MASTER'S THESIS

Model-based whitebox fuzzing REST web services to detect vulnerabilities

Kleuskens, J.

Award date:
2021

MODEL-BASED WHITEBOX FUZZING REST WEB SERVICES TO DETECT VULNERABILITIES

by

Jason Kleuskens

in partial fulfilment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University of the Netherlands, Faculty of Science
Master's Programme in Software Engineering
to be defended publicly on Thursday, July 8, 2021, at 2:00 PM.

Student number:		
Course code:	IM9906	
Education:	Master of Science - Software Engineering	
Date:	2021-07-02	
Document:	Graduation thesis	
Thesis committee:	dr. ir. H. P. E. Vranken (chairman),	Open University
	dr. ir. H. P. E. Vranken (supervisor),	Open University
	dr. G. Alpár (supervisor),	Open University

Author Student ID number	Jason (J.O.) Kleuskens
Title	Model-based whitebox fuzzing REST web services to detect vulnerabilities
Date of presentation Degree programme	2021-07-08 14:00 Open University of the Netherlands, Faculty of Science, Master's Programme in Software Engineering
Graduation committee Chair Primary supervisor Secondary supervisor	 dr. ir. H. P. E. Vranken dr. ir. H. P. E. Vranken dr. G. Alpár
Course code	IM9906

ACKNOWLEDGEMENTS

Although I enjoyed obtaining new knowledge on software security, at this point, I am also pleased to finalise this thesis. During this research, I discovered points on which I still have to work in the future to become better. One of them is a more critical attitude toward things. I hope the future will enable me to improve myself.

I would not have been able to realise this thesis without the feedback and support of several people. Therefore, I would like to thank several people.

At first, I would like to thank my supervisors, Harald Vranken and Greg Aplár. Thank you for letting me perform this thesis under your supervision. Furthermore, I would like to thank you for your time and good feedback to improve my thesis.

Secondly, I would like to thank my friends and family. Without their support, I would have never started and finished this journey. Mainly, I want to thank Tamara. Days after I worked late into the night on my thesis, you were always there for me the next day. Without your support, this journey would have been way tougher. My appreciation for everything you do for me is indescribable.

Finally, I would like to thank my employer, JNA Sportdrinks B.V. Thank you for funding my study and providing the necessary resources where they were needed.

Contents

SUMMARY

Currently, the growing number of REST web services on the Internet exposes more vulnerabilities that attackers can exploit. Connectedness, extensibility, and the increasing complexity of systems are three of the main reasons why REST web services become more vulnerable to these exploits.

Prior research shows that the model-based whitebox fuzzing approach is more effective on program binaries when comparing it with the model-based blackbox and traditional whitebox approach. Furthermore, research has proven that model-based fuzzing is effective on REST web services. Within this thesis, a model-based whitebox fuzzer prototype is developed. With this prototype, it is possible to answer the research question if the model-based whitebox fuzzing approach is more effective than a model-based blackbox approach and a traditional whitebox fuzzing approach on REST web services.

The use of empirical and quantitative research helps to answer this question. The research starts with studies in different areas. The first study shows the vulnerabilities in REST web services that are fuzzable for a model-based whitebox fuzzer. The results show that the vulnerabilities that are fuzzable for the model-based whitebox fuzzer in REST web services are HTTPS, access control, injection, validating content types, error handling and exposing sensitive information in HTTP requests. A second study explains how this research can use the model-based whitebox fuzzing approach from prior research for the model-based whitebox fuzzer for REST web services. The results show that the prototype can use the same processes as in the original approach. The methods within the fuzzing approach for creating the test inputs are target selection, file cracking, file stitching and file repair. Some changes have been made to use the OpenAPI specification instead of program binaries, but a similar approach is achievable. The last study shows the metrics that need to be used to compare fuzzing methods. These metrics are code coverage, valid requests, executed tests, and the number of revealed vulnerabilities. This research uses these metrics to ultimately analyse the performed experiments.

The model-based whitebox fuzzer prototype outperforms the traditional whitebox fuzzer and the model-based blackbox fuzzer when running tests with WooCommerce and Drupal as test programs. The results show that the model-based whitebox fuzzing prototype is overall more effective than the other two fuzzers. The prototype achieved a higher code coverage and generated more valid requests on both target programs. An explanation for these results is that the prototype is stateful and uses previous responses to create new inputs to reach deeper within the SUT. By contrast, the traditional whitebox fuzzer creates only valid requests for WooCommerce but could not perform a valid request on Drupal. During the experiments, several vulnerabilities are discovered, e.g., full path disclosure and HTTPS. However, none of these vulnerabilities is new. They were discovered and patched in the past. This may be due to the large, well-developed testing programs this research uses. Newer projects may be more about undetected vulnerabilities because a smaller group is working on this. Future research could add other techniques to extend the model-based whitebox fuzzer. For example, source code analysis to make it possible to detect more vulnerability types.

SAMENVATTING

Momenteel legt het groeiende aantal REST webservices op internet meer kwetsbaarheden bloot die aanvallers kunnen misbruiken. Verbondenheid, uitbreidbaarheid en de toenemende complexiteit van systemen zijn drie van de belangrijkste redenen waarom REST-webservices kwetsbaarder worden voor deze aanvallen.

Eerder onderzoek toont aan dat de op modellen gebaseerde whitebox fuzzing aanpak effectiever is voor programma-binaire bestanden in vergelijking met de modelgebaseerde blackbox en traditionele whitebox aanpak. Bovendien heeft onderzoek aangetoond dat modelgebaseerd fuzzen effectief is op REST-webservices. Binnen dit proefschrift wordt een modelgebaseerd whitebox fuzzer prototype ontwikkeld. Met dit prototype is het mogelijk om de onderzoeksvraag te beantwoorden of de modelgebaseerde whitebox fuzzing aanpak effectiever is dan een modelgebaseerde blackbox aanpak en een traditionele whitebox fuzzing aanpak op REST webservices.

Het gebruik van empirisch en kwantitatief onderzoek helpt om deze vraag te beantwoorden. Het onderzoek begint met onderzoeken op verschillende gebieden. De eerste studie toont de kwetsbaarheden in REST webservices die fuzzable zijn voor een modelgebaseerde whitebox fuzzer. De resultaten laten zien dat de kwetsbaarheden die fuzzable zijn voor de modelgebaseerde whitebox fuzzer in REST webservices zijn: HTTPS, toegangscontrole, injectie, valideren van inhoudstypen, foutafhandeling en het blootleggen van gevoelige informatie in HTTP verzoeken. Een tweede studie legt uit hoe dit onderzoek gebruik kan maken van de modelgebaseerde whitebox fuzzing aanpak uit eerder onderzoek voor de modelgebaseerde whitebox fuzzer voor REST webservices. De resultaten laten zien dat het prototype dezelfde processen kan gebruiken als in de oorspronkelijke aanpak. De methoden binnen de fuzzing aanpak voor het maken van de testinvoer zijn de doellocatie bepalen, het opbreken van invoeren, het samenvoegen van invoeren en het herstellen van invoeren. Er zijn enkele wijzigingen aangebracht om de OpenAPI specificatie te gebruiken in plaats van programma-binaire bestanden, maar een vergelijkbare aanpak is toegepast. De laatste studie toont de meetwaarden die moeten worden gebruikt om fuzzing methoden te vergelijken. Deze meetwaarden zijn codedekking, geldige verzoeken, uitgevoerde tests en het aantal kwetsbaarheden dat is onthuld. Deze meetwaarden zijn uiteindelijk gebruikt om de uitgevoerden experimenten te analyseren.

Het modelgebaseerde whitebox fuzzer prototype presteert beter dan de traditionele whitebox fuzzer en de modelgebaseerde blackbox fuzzer bij het uitvoeren van tests op WooCommerce en Drupal als testprogramma's. De resultaten laten zien dat het modelgebaseerde whitebox fuzzing prototype over het algemeen effectiever is dan de andere twee fuzzers. Het prototype bereikte een hogere codedekking en genereerde meer geldige verzoeken op beide testprogramma's. Een verklaring voor deze resultaten is dat het prototype stateful is en eerdere reacties gebruikt om nieuwe inputs te creëren om dieper in de functies van testprogramma's te komen. De traditionele whitebox fuzzer creëert alleen geldige verzoeken voor WooCommerce, maar kon geen geldig verzoek uitvoeren op Drupal. Tijdens de experimenten worden verschillende kwetsbaarheden ontdekt, bijvoorbeeld volledige

padonthulling en HTTPS. Geen van deze kwetsbaarheden is echter nieuw. Ze zijn in het verleden ontdekt en verholpen. Dit kan te wijten zijn aan de grote, goed ontwikkelde programma's die dit onderzoek gebruikt. Nieuwere projecten beschikken wellicht vaker over ongedetecteerde kwetsbaarheden, omdat hier een kleinere groep mee bezig is. Toekomstig onderzoek zou andere technieken kunnen toevoegen om de op modellen gebaseerde whitebox fuzzer uit te breiden. Bijvoorbeeld broncode analyse om meer kwetsbaarheidstypen te kunnen detecteren.

1

INTRODUCTION

The Internet plays a central role in the world today. However, the growing number of available web services and applications on the Internet exposes more bugs and vulnerabilities for people with bad intentions. A bug is when the system is not behaving as it is supposed to, whereas a vulnerability is a bug that manifests itself as an opportunity for exploitation [21]. An increasing number of vulnerabilities in web services is the result of this.

When attackers exploit these vulnerabilities, the exploitations of these vulnerabilities could lead to severe damage. In addition, various threats to confidentiality and integrity exist, which means that confidential information is accessible and unauthorised actors could taint this information. In terms of web services, these web services could leak account information or send false data to the web service, which affects the system or others. This means that even though web services can be beneficial, they also provide a risk. These issues have evolved; connectedness, extensibility, and complexity together contribute to how problems have evolved [37].

Firstly, there are more computer systems connected to the Internet. Initially, traditional computers and laptops are connected to the Internet. This was followed by connecting mobile devices such as smartphones and tablets. Nowadays, the Internet-of-Things (IoT) and Wireless Sensor Networks (WSNs) [18] used for customer products like televisions and webcams connected to the Internet. Connectedness is the term that stands for our society increasingly depending on Internet communication. Unfortunately, as systems connect to the Internet, they become more vulnerable to attacks. This is because attackers no longer need physical access to a system to exploit it. In addition, some of the existing systems never intended to connect to the Internet. This results in systems that do not have or lack built-in security measures[37].

Secondly, extensibility implies that software can update or extend[37]. To better understand this term and add more context to it regarding web services, web services can be extended with additional endpoints or updated with a new .NET [1] version. Extensibility is great from functional and economic viewpoints. Extensibility is also great for distributing patches for software applications to improve software security. However, preventing systems against vulnerabilities becomes challenging when added extensions or updates introduce the vulnerabilities. This could happen when the extensions become more complex. These vulnerabilities are adopted and are most often only fixable for the supplier of

[1] https://dotnet.microsoft.com/

the extension. Users remain dependent on the suppliers of these extensions/updates. For example, if the extension uses a system interface and the system does not seal it properly by default. Then this vulnerability is exploitable until the extension supplier seals it properly.

Thirdly, software systems have increased in size and complexity significantly. The security levels of the underlying systems and third-party components impact the security levels of the web service itself. As the complexity of these systems and third-party components increases, so does the chance for vulnerabilities within them[37]. This vulnerability could potentially open up a vulnerability for the web service.

Vulnerabilities are available for all types of web services. This includes Representational State Transfer [18] (REST) web services. REST is a software architectural style. Most often, a web service uses the REST architecture, and it uses a subset of HTTP. The REST architecture builds on a client-server communication pattern. A set of design principles (see paragraph 2.1) indicates how to correctly apply the REST software architectural style. Within these design principles, addressability, statelessness, and uniform interface are the most characteristic.

There are different methods to document the specification of the REST web services. All of these methods have in common that they are understandable for both humans as machines. This makes it possible for machines to understand what is possible to do with the REST web service and show the possibilities to human readers.

The Open Web Application Security Project (OWASP)[2] is an organisation that works to improve software security. This party frequently makes a list of the most occurring threats to web services and other software projects. Within those lists, it is easy to see which vulnerabilities attackers are most often trying to target. The OWASP also monitors REST security. The OWASP provides a top 10 vulnerabilities [4] and a penetration test strategy [6] for REST web services.

There are several methods to discover vulnerabilities within programs. Fuzzing is one of them. Fuzzing is an effective and widely used technique for finding vulnerabilities in software [11]. Using this technique could help to discover a vulnerability before an attacker does and perhaps exploits it. The fuzzing tool, fuzzer, create inputs and sends these to the system under test (SUT). This technique may trigger a vulnerability inside the system. Most often, the fuzzer reports the vulnerability to the executor of the fuzzer. The inputs used can be syntactically or semantically incorrect. The fuzzer monitors the behaviour and response of the SUT when the created input is processed. While the fuzzer monitors the SUT, vulnerabilities can occur, and the fuzzer logs them.

Currently, there are three types to categorise a fuzzer. A fuzzer can be a blackbox, a greybox or a whitebox. The level of program understanding should be analysed to place a fuzzer within a group. A whitebox fuzzer almost has a complete program understanding of the SUT, and a blackbox fuzzer does not know what is inside the SUT. The greybox fuzzer falls between those two. Based on the level of program understanding, it leans more to whitebox or blackbox.

There are different methods to generate the input that the fuzzer uses during the fuzzing process. There are two approaches for input generation. It can be a mutation-based method or a generation-based approach. In the case of mutation-based, mutations are performed on an initial seed to create new inputs. For the generation-based approach, block-, model- or grammar-based approaches are available. With those options, the fuzzer can generate

[2] https://owasp.org/

inputs that follow a pattern. Usually, this results in more inputs that are valid for the SUT.

Currently, no research combines model-based whitebox fuzzing with REST web services. Since REST is a frequently used architecture, it is worth investigating this fuzzing method on REST web services. This research combines some techniques to validate if model-based whitebox fuzzing works for detecting vulnerabilities in REST web services.

The research of Pham et al.[32] shows that model-based whitebox fuzzing is a more effective method for fuzzing project binaries than model-based blackbox fuzzing and traditional whitebox fuzzing. This research aims to validate if model-based whitebox fuzzing is a more effective method than model-based blackbox fuzzing and traditional whitebox fuzzing when applied on a REST web service. A prototype is developed to reach this goal, but first, a study covers how the approach of Pham et al.[32] could be used on REST web services. The developed prototype performs several tests on different systems under test (SUTs). Additionally, performance metrics should be studied to compare fuzzing systems. Comparing the performance metrics of the model-based whitebox fuzzer with the model-based blackbox fuzzer and traditional whitebox fuzzer gives more insight into how effective it is.

The contributions of this work are as follows:

- A proposal for a model-based whitebox approach that is capable of detecting vulnerabilities in REST web services.
- A prototype for model-based whitebox fuzzing REST web services based on the proposed approach.
- An evaluation of the effectiveness of the proposed approach based on metrics. This evaluation uses several applications with REST web service as a SUT.

Besides validating a model-based whitebox fuzzing approach on REST web services, this research provides a new fuzzing approach for REST web services. Furthermore, this approach is built explicitly for REST web services instead of program binaries. In addition, this approach is based on the specification of REST web services. This specification is readable for both humans and machines. Program binaries are readable for machines, binaries are written in machine language. This makes the proposed approach easier to understand for humans. Possibly, future research could validate this approach on other types of projects or extend the prototype with other fuzzing techniques on REST web services.

The remainder of the thesis covers several subjects. Chapter 2, background, contains more and more detailed background information. This background information covers the topics fuzzing, REST, model-based testing and model-based whitebox fuzzing. Chapter 3, research, covers the research questions, chosen research method and an explanation of how the prototype is tested. Chapter 4, literature study, consists of two studies. The first study covers the vulnerabilities that are available for REST web services. Also, for each vulnerability, it is mentioned how the developers can prevent the vulnerability and if the vulnerability is fuzzable. The last study covers the different available metrics to compare fuzzing methods. Chapter 5 explains how the model-based whitebox fuzzer prototype is eventually built during this thesis. This also covers how the methods used in model-based blackbox and traditional whitebox fuzzing can be combined to create a model-based whitebox fuzzer for REST web services. Finally, this chapter also contains the results of the experiments. Also, the implementation of the prototype is described together with the results of

the experiments. Chapter 6, discussion, conclusion and future work, include the answers of this research to the research questions. Also, this chapter discusses directions for further research on this subject. Finally, chapter 7, reflection, contains a reflection on the actions taken during the research.

2

BACKGROUND INFORMATION

Prior research provides background information that helps to understand the used techniques within this thesis. Therefore, this chapter describes those techniques. This chapter covers REST web services, fuzzing in general, model-based testing, and model-based whitebox fuzzing.

2.1. REST WEB SERVICES

Representational State Transfer (REST) is a description of an architectural style. The REST architecture is most often used for web services. The REST architecture results in client-server communication. This means that the client sends a request to the server, the server processes these messages and returns responses. REST messages most likely consist of an XML or JSON format and are transported using HTTP. Figure 2.1 presents an overview of the communication flow.

Figure 2.1: REST communication adapted from Phpflow.com[1]

Design principles of REST include addressability, statelessness, and uniform interface [29]. Additionally, client-server, cache, layered system, and code-on-demand are other design principles use within REST.

- **Client-Server** In a client-server design, the client sends a request to the server. The server processes the request and returns a response to the client. The separation of

concerns is the reason behind this design principle[13]. This improves the portability of the user interface and improves scalability by simplifying the server components.

- **Addressability** With addressability, REST models the datasets to operate on and presents these datasets as resources.
- **Uniform interface** REST uses uniform interfaces to access the REST resources using a fixed set of Hypertext Transfer Protocol (HTTP) methods. Typically, REST web services use the GET, PUT, POST, and DELETE HTTP methods to retrieve the create, read, update and delete (CRUD) functionality.
- **Statelessness** Furthermore, REST builds on the statelessness principle. This means that every transaction between client and server is independent and unrelated to previous transactions between client and server. The server does not store any state information of the client. When the client needs to have a state, it is self-responsible to keep track of it.
- **Cache** The cache design principle is applied to REST to improve network efficiency. The cache design principle requires that the data within the response is implicitly or explicitly labelled as cacheable or non-cacheable. When a response is cacheable, the server granted the client the right to reuse that response data for later, equivalent requests[13]. This takes away the latency between the client's request and the server's response.
- **Layered System** The layered system design patterns allow architecture to be composed of hierarchical layers. Each component's behaviour could have a constraint that limits its range. Bounds can be added around each layer which is also beneficial for security. The disadvantage is that a layered system adds overhead and latency to the data processing [13].
- **Code-On-Demand** Code-On-Demand is the last design principle. This principle is optional since it reduces the extensibility of the REST service. Code-On-Demand enables web servers to transfer executable programs to clients temporarily [13].

Description Specification Many techniques are available to document the specification of a REST web services with its endpoints and resources to readers. Examples of techniques are OpenAPI [2], JSON Schema [3], RAML [4] and WSDL [5]. These specifications contain information about Uniform resource identifiers (URIs) and other resources, endpoints, HTTP methods supported by the endpoints, the expected input models, and the expected HTTP response codes and response model. From all these techniques, the OpenAPI technique for documenting REST web services is the most popular option.

The description specification is most often good and easy to read for readers. However, these readers are not only human but also and mainly computers. When the specification is readable by computers, it enables computers to show the information nicely for the users.

[2] https://swagger.io/specification/
[3] https://json-schema.org/
[4] https://github.com/raml-org/raml-spec
[5] https://www.w3schools.com/xml/xml_wsdl.asp

For example, the OpenAPI scheme is machine-readable, and tooling like Swagger [6] can process it.

2.2. FUZZING

Fuzzing is an effective and widely used technique for finding vulnerabilities in software [11]. Using this technique could help to discover a vulnerability before an attacker does and perhaps exploits the vulnerability. Where developers and system engineers can use fuzzers to act defensive, attackers can use the same kind of fuzzers to act offensive. During fuzzing, the fuzzer runs a SUT. The fuzzer will use an input generation strategy to send inputs to this SUT. These inputs could be valid or invalid. When they are invalid, they are syntactically or semantically incorrect. In those cases, the SUT will most likely deny them. The fuzzer monitors the behaviour and response of the SUT when the created input is processed. While the fuzzer monitors the SUT, vulnerabilities can occur, and the fuzzer logs them. Many companies such as Microsoft [3] and Google [2] [1] have integrated fuzzing within their software development lifecycle. Security auditors and open-source developers have followed this trend, and both have started to use fuzzing to provide and assure end-users with secure software[26].

Initially, Professor Miller started with fuzzing [28]. Miller wanted to test the robustness of various UNIX utilities [7], given an unpredictable input. During the research, Miller created an input generator to create a stream of random characters and used these inputs to attack as many utilities as possible. Within the performed experiments, 25-33% of the utility programs on any UNIX versions that Miller tested did crash[28].

TERMINOLOGY

The fuzzing terminology is introduced based on the article from Chen et al. [11]. This research introduces this terminology because the fuzzing community is vibrant. The number of fuzz testing related articles is rapidly increasing, as displayed in figure 2.2, which displays new (IEEE [8]) fuzz testing related articles for each year between 1989 and 2020. Different terms describe the same technique or a similar term for different techniques between those articles. For example, various terms define the target program, e.g. System Under Test (SUT), Program Under Test (PUT), Unit Under Test (UUT). Within this research, only SUT is the used synonym for the target program.

For this reason, this research selects one standard. This standard is based on the definitions mentioned by Chen et al. [11]. This research describes some of the definitions below. This research uses this standard throughout this thesis.

Fuzzing Fuzzing is a highly effective vulnerability detection technique. It tests a system with the continuous processing of test cases generated by another program. At the same time, the fuzzer monitors the SUT to expose any defects revealed by processing this input [11].

Fuzz testing Fuzz testing is the use of fuzzing to test a SUT [26]. The goal is to see if the SUT crashes or succeeds and exposes a vulnerability in the meantime. Historically, fuzz

[6] https://editor.swagger.io/
[7] http://parallel.vub.ac.be/documentation/linux/unixdoc_download/Utilities.html
[8] https://www.ieee.org/

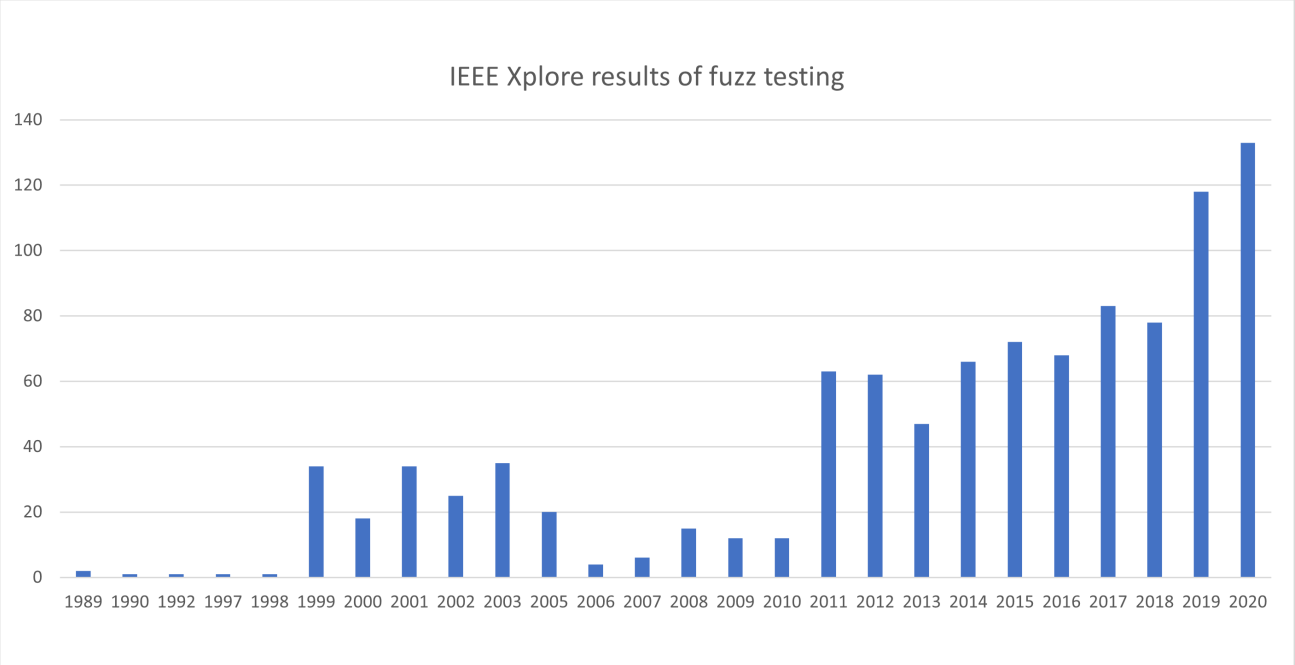

Figure 2.2: Results of fuzz testing articles from IEEE Xplore

testing is mainly used to find security vulnerabilities [11].

Fuzzer A fuzzer is a program that performs fuzz testing on the SUT [26]. The fuzzer consists of different modules to handle different tasks[11]. A fuzzer makes fuzzing possible on SUTs.

Anatomy of a fuzzing system

The original structure of the early fuzzing systems is quite simple. The blue dashed frame in figure 2.3 shows an example of a fuzzing system. The system contains five modules: the test case generator, the delivery module, the target program, the bug detector and the bug filter. The black arrows visualise the data flow between the modules that transfer data, e.g., the test case generator transfers the test cases to the delivery module, and the bug detector transfers bugs to the bug filter. Within the same figure, red arrows visualise the relation between modules that control each other, e.g., the monitor monitors the target program and the static analyzer to provide runtime and static information [11].

The test case generator creates an input for the SUT; this input represents one test case. The generator can use different strategies for input generation to improve the efficiency of fuzzing. One of the strategies that the fuzzer can use is random mutating. In this case, the initial input seed is getting mutated for each test case.

The delivery module receives test cases from the test case generator and delivers them to the target program for execution [11].

The target program is the system that is under test, also known as SUT. It can be all types of systems, e.g., binary code (with or without source code), a web service, an operation system, a compiler.

The bug detector collects and analyses relevant information when a target program crashes and reports errors in a test case. The bug filter is usually done manually and, therefore, time-consuming. However, some tools have built a bug filter on top of the fuzzer.

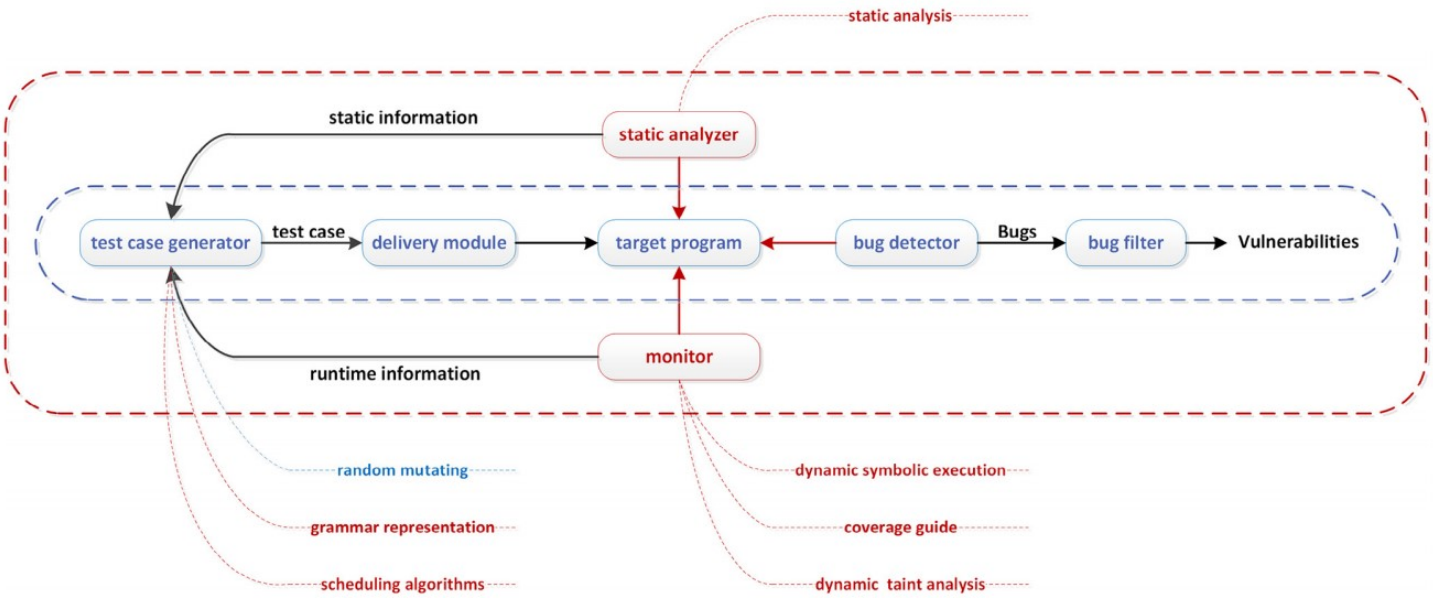

Figure 2.3: Fuzzing system architecture, adopted from Chen et al. [11]

Modifications

The early fuzzing system has been found inefficient for modern applications. Most modern applications have more complex input structures and are larger than the older Unix applications. Researchers have introduced different techniques to improve the efficiency of fuzzers. The techniques include grammar representation [14], scheduling algorithms [9], static analysis [22], dynamic symbolic execution [10], coverage guide/feedback [9], dynamic taint analysis [31], and machine learning [17]. The red dashed frame in figure 2.3 includes these techniques.

The monitor obtains runtime information, which the monitor sends to the test generator. The runtime information sent to the test generator includes the symbolic expressions, coverage data, and dynamic taint analysis gathered from the added techniques.

The static analyser performs a static analysis. During this process, the static analyser extracts static information from a binary program or source code to find potential bugs and direct the fuzzing process [8]. The static information includes control flows, potentially vulnerable code, and specified patterns.

While the fuzzing process takes place, the fuzzer performs the dynamic analysis. During this process, the fuzzer updates the test case generator with feedback received from the SUT. This makes it possible for the test case generator to generate better inputs.

Basic Taxonomy of Fuzzer types

Three types categorise a fuzzer: whitebox, blackbox and greybox fuzzing [11]. The most significant difference between these groups is the amount of gathered information, e.g. program understanding, about the SUT.

Blackbox fuzzer

When using a blackbox fuzzer, the fuzzer does not look at the logic inside of the SUT. The fuzzer continuously provides input data and observes the output results from the SUT. Black-box fuzzing is at one of the extremes in terms of the level of program understanding.

Whitebox fuzzer

At the other extreme is whitebox fuzzing. In this case, the fuzzer can obtain detailed information on the SUT. Whitebox fuzzing can obtain information on the SUT, e.g., source code and design specifications. The fuzzing process utilises this information to improve the efficiency of the fuzzing process. When the fuzzer knows that the program does not respond to an input, it makes no sense in trying it again.

Greybox fuzzer

Between these two extremes lies greybox fuzzing. Greybox fuzzing obtains some information about the SUT. However, it differs from whitebox fuzzing since greybox fuzzing is not built on heavyweight analysis and constraint solving. Also, in some cases, the greybox fuzzer can obtain the code coverage to reach deeper into the SUT. What is possible with a fuzzer depends on the design of the fuzzer. For example, a greybox fuzzer like AFL[9] leverages the code coverage to reach deeper into the SUT. When the whitebox fuzzer requires the user to provide much information, a greybox fuzzer potentially does not require all this information. In those cases, a greybox fuzzer could be quicker to set up than a whitebox fuzzer. On the other hand, a blackbox fuzzer does not need any information about the SUT. This makes the greybox fuzzer potentially slower to set up than the blackbox fuzzer.

INPUT GENERATION STRATEGIES

According to the input generation strategy, fuzzing has two categories: mutation-based and generation-based. The mutation-based approach uses a prepared input seed. The fuzzer performs random transformations on this seed. For example, if the seed contains the value 'fuzzing', the transformations could look like 'fazzing', 'fussing' or 'flaspnn'. In the generation-based approach, the fuzzer generates inputs using a formal, semi-formal or informal input specification, e.g., a model, a block or a grammar [11]. When using a model that can create a JSON object, the model knows which properties within the JSON object are fuzzable. For example, for a JSON object with two properties, id (int) and name (string), the fuzzer generates the inputs for those properties. Finally, the generated inputs from the generation-based fuzzer could look like these: 'id': 1, 'name': 'fuzzing', 'id': 12, 'name': 'model-based' or 'id': 123, 'name': 'hello world'.

In the case of mutation-based fuzzing, the target program can reject the inputs early during processing. This is possible when the input data varies too much from the format expected by the target program. In the case of generation-based fuzzing, this problem is relieved by the input data specification. Therefore, mutation-based fuzzing almost always generates a lower code coverage than generation-based fuzzing. However, the creation of the input data specification is often very time-consuming. Figure 2.4 compares the two types of fuzzing approaches[24].

2.3. MODEL-BASED TESTING

Model-based testing has an advantage over mutation-based testing. The advantage of model-based testing is that it uses its knowledge about the message structure to systematically generate messages containing invalid data among valid data [34]. This makes it possible to generate more valid requests that could target more components within the software. For a mutation-based input, the chance to reveal a vulnerability is lower [34].

[9] https://lcamtuf.coredump.cx/afl/

	Easy to start ?	Priori knowledge	Coverage	Ability to pass validation
Generation based	hard	needed, hard to acquire	high	strong
Mutation based	easy	not needed	low, affected by initial inputs	weak

Figure 2.4: Comparison of generation-based and mutation-based fuzzers adapted from Li et al. [26].

In model-based testing, the fuzzer generates test cases based on a model-based approach [34]. Within a model-based approach, the model represents a simple, abstract version of the SUT. Model-based testing uses models from three groups: formal, semi-formal, and informal models [30]. Formal models have a mathematical foundation. The use of formal models is problematic because scaling requires many resources. For informal models, the absence of a uniform definition makes it difficult to model the complexity of modern applications. This results in that most of the models are semi-formal. These models can not prove properties mathematically, but these models make it possible to allow structured, automated testing of complex applications [15].

Since semi-formal models are the most common, this paragraph explains what a semi-formal model should look like for REST web services. The semi-formal model should contain three components: an interface description, a behavioural model, and deployment information to perform automated model-based testing [20]. The interface description describes the available interfaces in a SUT. Furthermore, the interface description describes the expected inputs and outputs. Based on this information, the fuzzer selects the target locations with their expected inputs. These target locations are the available interfaces. For REST web services, the target locations are the paths with their HTTP method (POST, GET, PUT, and DELETE). The behavioural model contains information about how the SUT behaves. Most important for REST web services are the dependencies between endpoints and the order in which these endpoints should be. The deployment information is the last part of the model. The deployment information describes the deployment location of the SUT. For REST web services, a Uniform Resource Locator (URL) to the location of the specification would be sufficient. Other information can be obtained based on this specification, e.g., the version, the host URL, and the HTTP method.

Model-based testing generates test cases based on the obtained model. Atlidakis et al. [7] use the OpenAPI specification of a REST web service to create the model. The OpenAPI specification contains the information for each endpoint. The model should extract the available endpoints, inputs, and outputs from this specification. The dependencies between endpoints should be derived from the OpenAPI specification to complete the model.

2.4. Model-based whitebox fuzzing

This research implements the model-based whitebox fuzzing technique. This subsection will focus on how prior research executes the model-based whitebox fuzzing approach.

There are two approaches available for model-based whitebox fuzzing after searching

model-based whitebox on Google scholar [10]. Letichevsky et al. introduce a model-based whitebox fuzzing approach based on symbolic execution [23]. This approach uses a formal model and generates traces based on the source code of the SUT. Concrete tests for deterministic systems testing result from checking the conformance between the actual outputs in the traces and the expected outputs generated by the model. Furthermore, Pham et al. introduce an approach for program binaries [32]. The second approach is described in more detail, includes multiple interesting techniques (including symbolic execution) to create the inputs and is more often referred to within fuzzing related papers. The remainder of this subsection analyses the second approach.

Pham et al. introduce a model-based whitebox fuzzing (MoWF) approach [32], which shows that this technique effectively finds security vulnerabilities within a SUT. With this approach, model-based whitebox fuzzing is a marriage of model-based blackbox fuzzing (MoBF) and traditional whitebox fuzzing (TWF). For model-based whitebox fuzzing, the model-based blackbox fuzzing generates the valid input files. This is done efficiently with the known objects within the model. The whitebox fuzzer processes the input files and monitors how the fuzzer behaves during the fuzzing process.

Model-based whitebox fuzzing follows four steps while fuzzing: Seed selection and file cracking, adding and removing data chunks, changing data fields in inserted data chunks, repeat. The model-based whitebox fuzzer selects an initial input closest to a potential crash location during seed selection and file cracking. The fuzzer uses static analysis from the Intel Dynamic Binary Instrumental Tool: Pin [25] to obtain potential crash locations. The fuzzer then tests these potential crash locations to validate the analysis and check if this leads to a vulnerability. The model-based whitebox fuzzer instantiates the input from the input model provided by the model-based blackbox fuzzer. Then, the model-based whitebox fuzzer marks all data fields which the user has specified as "modifiable". Model-based whitebox fuzzer considers only these marked data fields for fuzzing. After the fuzzer creates the input file, the fuzzer creates a pool of input fragments. This process is called file cracking. Within this process, the fuzzer considers all other files as potential donor files; these are files from which the fuzzer uses fragments to make a new whole. The fuzzer disassembles the donor files and adds the file fragments to the pool of input fragments.

In the second step, adding and removing data chunks, fragments of data are added and removed from the input. This is necessary because the SUT only executes some inputs if a specific data chunk is absent or present within the input. In REST web services, these data chunks could be authentication keys in the header or essential properties within the request's body, e.g., an id. Once model-based whitebox fuzzing identifies the type corresponding to the data chunk being removed or added, the file stitcher coordinates the data chunk transplantation. The data chunk transplantation is responsible for placing the correct values for the parameters within the input file. This process has two steps. First, the file stitcher searches the fragment pool for candidate fragments. Finally, the file stitcher uses the input model to transplant them into the appropriate locations in the input. The goal of this step is to create a valid input file for the target location.

In the third step, the fuzzer modifies the data fields within the valid input file. A method symbolic execution is used to get valid inputs for the data fields based on the model. At this point, the difference with model-based blackbox fuzzing is made. Deep vulnerabilities that require specific values are best exposed by a symbolic execution-based approach [32]

[10] https://scholar.google.com/scholar?hl=nl&as_sdt=0%2C5&q=model-based+whitebox&btnG=

because when the fuzzer generates more valid inputs, the chance of hitting a deep vulnerability is more significant. When the file is complete, the fuzzer performs checks to confirm that the inputs are valid. If this is the case, the fuzzer sends the inputs to the target location and are processed there. Once the input reaches the target location, the fuzzer monitors the SUT and checks if a crash occurs.

In the last step, repeat, data chunks can be nested. Model-based whitebox fuzzing uses these generated nested files as new seeds to continue the next iteration, starting from step 1. From the initial seeds and the new seeds, model-based whitebox fuzzing selects the location closest to the previous location and performs the steps again to create new inputs.

Listing 2.1 shows the entire model-based whitebox fuzzing approach. Lines 1 until 7 are performed with static analysis to discover the available targets and inputs. The previous steps cover the lines from 8 until 24.

Listing 2.1: Model-based whitebox fuzzing algorithm from Pham et al. [32]

```
Input: Program P, Input Model M
Input: Initial Test Suite T, Targets L
Output: Augmented Test Suite T'
1: if L = ∅ then
2:     L ← IDENTIFYCRITICALLOCATIONS(P)
3: end if
4: if T = ∅ then
5:     t ← INSTANTIATEASVALIDINPUT(M)
6:     T ← {t}
7: end if
8: while timeout not exceeded do
9:     Target location l ← CHOOSETARGET(L)
10:    Input file t ← CHOOSEBEST(M)
11:    Fragment Pool Φ ← FILECRACKER(T,M)
12:    Crucial IFS Λ ← DETECTCRUCIALIFS(t, l, P, M)
13:    for all λ ∈ Λ do
14:        Valid files T_λ ← FILESTITCHER(t, λ, Φ,M)
15:        for all t_λ that negate λ do
16:            Hybrid file t''_λ ← MARKSYMBOLICVARS(t_λ,M)
17:            Files F ← PATHEXPLORATION(t''_λ, λ, l, L, P)
18:            for all f ∈ F do
19:                Valid file f' ← FILEREPAIR(f,M)
20:                T ← T ∪ f'
21:            end for
22:        end for
23:    end for
24: end while
25: T' ← T
```

Pham et al. test this approach with an experiment [32]. The selected subjects are from a pool of well-known program binaries of video players, document readers, music players, and image editors within this experiment. In total, eight programs were chosen, includ-

ing more well-known program binaries like Video Lan Client (VLC) [11] and Windows Media Player (WMP) [12]. The model-based blackbox fuzzer, Peach [13], enables the model-based whitebox fuzzer to create the inputs. The traditional whitebox fuzzer, Hercules [33], a traditional whitebox fuzzer for program binaries, eventually performed the tests.

Eventually, the results show that the model-based whitebox fuzzer performed better than the traditional whitebox fuzzer and the model-based blackbox fuzzer individually. The different fuzzers are compared based on the discovered vulnerabilities in the tested systems. Overall, blackbox fuzzing approaches have known limitations. For conditional statements, the chance of being exercised could be slim. In addition, generation-based testing usually provides a higher code coverage compared to mutation-based testing [16]. These could be explanations for the performance compared to the model-based blackbox fuzzer and the traditional whitebox fuzzer.

[11] https://www.videolan.org/vlc/

[12] https://support.microsoft.com/nl-nl/windows/windows-media-player-12-e8f84f54-cd64-865c-2e83-1d8ec121b5b8

[13] https://github.com/MozillaSecurity/peach

3

RESEARCH

This thesis describes a research that builds on the studies of Gerritsen [15] and Pham et al. [32] by applying the model-based whitebox fuzzing approach on REST web services. This part of the thesis will cover the used research questions and research method.

3.1. RESEARCH QUESTIONS

This thesis is based on an existing model-based whitebox approach [32], which is validated on program binaries. Besides this research, only one other study has been found that proposes a different approach for model-based whitebox fuzzing. However, this approach only used one technique that is also available within the approach from Pham et al. [32]. This thesis aims to determine if a similar approach can be applied to REST web services. The main research question of this thesis is: **How can model-based whitebox fuzzing be applied to REST web services to detect vulnerabilities?** The following sub-questions are formulated to answer the question:

- **RQ1: What types of vulnerabilities can be detected in REST web services with model-based whitebox fuzz testing?**
  When it is clear which vulnerabilities often occur within REST web services, it is also possible to investigate whether the chosen fuzzing technique can expose them within a SUT.

- **RQ2: How can the model-based whitebox fuzzing approach on program binaries be applied to REST web services?**
  The goal is to use an existing model-based whitebox fuzzing approach for a prototype that targets REST web services. It is necessary to examine whether the existing approaches cover this, or where deviations must occur and why deviations must be made.

- **RQ3: How does model-based whitebox fuzzing perform on REST web services compared to traditional whitebox fuzzing and model-based blackbox fuzzing?**
  The approach for model-based whitebox fuzzing should be tested against other fuzzing techniques. In this research, model-based blackbox fuzzing and traditional whitebox

fuzzing are used to compare the effectiveness of the different fuzzing approaches. Experiments must be carried out to assess whether the model-based whitebox fuzzing approach is more effective for REST web services.

3.2. Research method

For the chosen research method, empirical research, a hypothesis can be created. The hypothesis is that a similar approach as used for model-based whitebox fuzzing on program binaries [32] also works on REST web services. Furthermore, better results are expected from model-based whitebox fuzzing on REST web services than from a traditional whitebox fuzzer or model-based blackbox fuzzing on REST web services. With a valid hypothesis, the main research question can be answered. A prototype is created that is capable of performing model-based whitebox fuzzing to support this research. Additionally, this research uses quantitative research by using multiple experiments to get more results for the comparison.

The research starts with a few studies in the following areas:

- **Vulnerabilities in REST web services** - This study covers RQ1. Based on the literature, this study finds the common vulnerabilities in REST web services. With the knowledge of REST web services and model-based whitebox fuzzing, discoverable vulnerabilities are selected.

- **Model-based whitebox fuzzing approach** - This study covers RQ2. This provides a better understanding of the existing model-based whitebox fuzzing approach. This research extends an existing model-based whitebox fuzzing approach with changes for fuzzing REST web services.

- **Metrics to compare fuzzing methods** - This study starts with RQ3. To eventually compare the different fuzzers, it is important to know which metrics should be measured. This study selects the usable metrics.

The results of the study lead to the implementation of a model-based whitebox fuzzer prototype. During the implementation, the fuzzer will automatically generate the model of the REST web service from the OpenAPI Specification (OAS)[1]. Previously, this specification was called the Swagger specification[7]. Based on this model, it is possible to target each input location of the SUT. The fuzzer has the potential to trigger vulnerabilities at these input locations. This model-based whitebox fuzzer is tested and validated against at least one system under test (SUT), but the aim is to use multiple SUTs. This would provide more data to make a better comparison between the fuzzers. This research compares the model-based whitebox fuzzer with a model-based blackbox fuzzer and a traditional whitebox fuzzer to determine the effectiveness of the model-based whitebox fuzzer. The comparison is made based on the metrics specified in the next chapter.

This research utilises publicly available SUTs. These SUTs are open source projects with REST web services. These projects are selected because they are open source, this means that all source code is available online for possible follow-up of the research.

[1] https://swagger.io/resources/open-api/

3.3. RESEARCH EXPERIMENTS

During the design, implementation, and validation phases, the prototype is tested against a testing SUT. Additionally, during the experiments, several SUTs are used. Section 3.3.1 describes the test SUT with known vulnerabilities. Section 3.3.2 covers the SUTs that are used during the experiments when the prototype has been completed. Section 3.3.3 describes the model-based blackbox fuzzer. This part includes the model-based blackbox fuzzer. The following section, section 3.3.4, describes the traditional whitebox fuzzer. Section 3.3.5 describes how combining both the fuzzers results in the model-based whitebox fuzzer.

3.3.1. TESTING APPLICATION

The testing SUT used during the design, implementation and validation phases of this prototype is called SutSqlI. SutSqlI is created by Arjan Gerritsen[15] and contains some SQL injection vulnerabilities. Within this SUT, no PreparedStatements were used to interact with the database. This flaw leads to a risk of SQL injection since PreparedStatements makes sure that unsafe characters are escaped. This SUT is a Spring Boot application written in Java and built on Maven. The application is extended with Swagger to make it possible to use the REST endpoints of the application. The SUT has four endpoints, as illustrated in Figure 3.1.

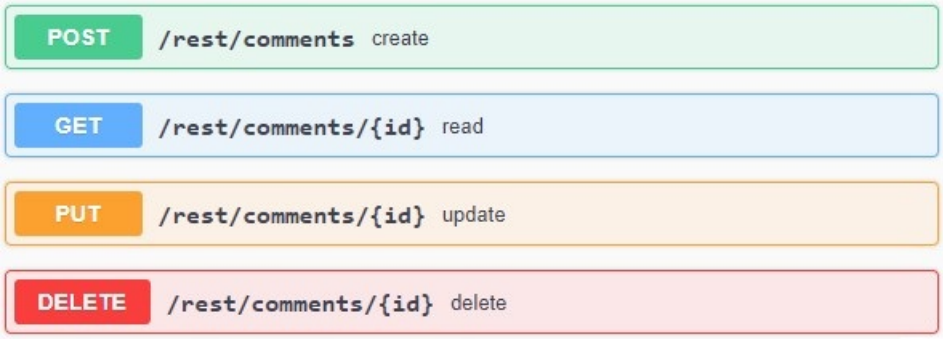

Figure 3.1: SutSqlI endpoints

Currently, SutSqlI is only built to detect SQL injections. Listing 3.1 displays the original source code of the endpoints. The implementation of the endpoints does not use PreparedStatements. The result of this is that unsafe payloads, e.g., "'", ";" and others [2], can escape the SQL query and perform malicious actions. For now, SutSqlI only contains SQL injections. A SQL injection with an unsafe payload could damage the SutSqlI system, e.g., delete the table or delete the database. However, SutSqlI has the potential to be extended and include multiple vulnerabilities.

Listing 3.1: SutSqlI REST endpoints implementation

```
@Service
public class CommentService {

    // variable(s)
    @Autowired
    private JdbcTemplate jdbcTemplate;
```

[2] https://github.com/payloadbox/sql-injection-payload-list

```
    public Comment create(Comment comment) {
        jdbcTemplate.execute("INSERT INTO comments (description)
            values ('"+ comment.getDescription() +"');");
        Long id = jdbcTemplate.queryForObject("SELECT LAST_INSERT_ID()
            ;", Long.class);
        comment.setId(id);

        return comment;
    }

    public Comment read(Long id) {
        Comment comment = jdbcTemplate.queryForObject("SELECT * FROM
            COMMENT WHERE ID = " + id + ";", Comment.class);
        return comment;
    }

    public Comment update(Comment comment) {
        jdbcTemplate.execute("UPDATE comments SET description = '"+
            comment.getDescription() +"' WHERE id = " + comment.getId()
             +";");
        return comment;
    }

    public void delete(Long id) {
        jdbcTemplate.execute("DELETE FROM comments WHERE id = " + id +
            ";");
    }
}
```

The original source code of SutSqlI contains a bug. For this reason, the function public Comment read(Long id); is rewritten. Both the SQL query and the use of the `queryForObject()` function are wrong. The SQL query selects data from "COMMENT", this table does not exist and should be "comments". The function returned an error (1 parameter expected, got 2). For this reason, the function is rewritten with the alternative `query()`. Applying both changes resulted in a working version of SutSqlI.

3.3.2. EXPERIMENT APPLICATIONS

The research will utilise publicly available SUTs. These SUTs are open source projects with REST web services. While exploring the Internet for fitting SUTs, the leading search query was 'open source REST web applications'.

The following open source projects are obtained and could potentially function as a

SUT: Magento Commerce[3], WooCommerce[4], Drupal[5], PrestaShop[6], MediaWiki[7], and Sylius[8].

These projects are selected because they are open source, this means that all source code is available online for possible follow-up of this research. Additionally, these projects are all built on PHP. Furthermore, when looking at vulnerability databases like the Common Vulnerabilities and Exposures (CVE) details [9], it is possible to see that these projects have a history with discovered vulnerabilities in the past.

SUT 1: Magento Commerce Looking at the CVE details of Magento[10], 30.7% of the vulnerabilities found are related to Cross-Site Scripting (XSS). Code Execution and SQL Injection have 30.2% and 4.5% of the vulnerabilities, respectively. In total, the CVE database contains 202 vulnerabilities until 2021 June 16th.

SUT 2: WooCommerce The CVE details of WooCommerce do not show much information[11]. The CVE database contains only 5 XSS, 1 Cross-Site Request Forgery and 1 Directory Traversal vulnerabilities for WooCommerce from 2017 until 2021 June 16th.

SUT 3: Drupal For the Drupal project, the CVE database contains 320 vulnerabilities [12] until 2021 June 16th. From these 320 vulnerabilities, 36.6% of the vulnerabilities are XSS related vulnerabilities.

SUT 4: PrestaShop PrestaShop has 58 discovered vulnerabilities in the CVE database between 2008 and 2021 June 16th [13]. Most of the vulnerabilities are XSS related vulnerabilities (41.4%). The percentage of vulnerabilities related to Code Execution and SQL Injection are 13.8% and 5.2%, respectively.

SUT 5: MediaWiki For MediaWiki, 245 reported vulnerabilities are visible between 2004 and 2021 June 16th [14]. Of all vulnerabilities, 38.0% are XSS related vulnerabilities, and 17.1% come from gaining information vulnerabilities.

SUT 6: Sylius For Sylius, the CVE database has only reported three vulnerabilities[15] until 2021 June 16th. The vulnerabilities cover two categories: 2x an XSS vulnerability and 1x a Gain Information vulnerability.

[3] https://magento.com/
[4] https://woocommerce.com/
[5] https://www.drupal.org/
[6] https://www.prestashop.com/en
[7] https://www.mediawiki.org/wiki/MediaWiki
[8] https://sylius.com/
[9] https://www.cvedetails.com/
[10] https://www.cvedetails.com/product/31613/Magento-Magento.html?vendor_id=15393
[11] https://www.cvedetails.com/product/35474/Woocommerce-Woocommerce.html?vendor_id=16011
[12] https://www.cvedetails.com/vendor/1367/Drupal.html
[13] https://www.cvedetails.com/product/15797/Prestashop-Prestashop.html?vendor_id=89507
[14] https://www.cvedetails.com/product/4125/Mediawiki-Mediawiki.html?vendor_id=2360
[15] https://www.cvedetails.com/product/73944/Sylius-Sylius.html?vendor_id=22179

EXPERIMENT ENVIRONMENT

A Virtual Machine (VM) executes the experiments during this research. A VM is to create systems that have the same specifications to run multiple experiments simultaneously. Important here is to make sure that all experiments have the same resources to compare them. This VM has the following specifications and tools installed:

- Operating System: Windows 10 (64-bit) Development Build 2104 Enterprise
- Base Memory: 8192 MB
- Processor(s): 5 CPUs
- Execution Cap: 100
- Video Memory: 128
- Installed tools:
  - XAMPP 64-bit 7.4.19.0
  - WooCommerce 5.3.0
  - WordPress 5.7.2
  - Drupal 9.1.10
  - Visual Code 1.55.2
  - PHP 7.4.19 with:
    - Zend OPCache 7.4.19
    - Xdebug 2.9.7.1

3.3.3. MODEL-BASED BLACKBOX FUZZING

The model-based fuzzer rest-fuzzer[15] implements the model-based blackbox fuzzer. This fuzzer uses the application programming interface (API) specification to generate a model. Based on this model, the fuzzer generates the entries and inputs. Rest-fuzzer uses principles based on principles from RESTler[7] for the fuzzer. For example, using the OpenAPI specification to generate the model comes from RESTler.

Rest-fuzzer checks for various bugs. With rest-fuzzer, when the fuzzer receives a response with status code 500 (Internal Server Error), a bug is reported. These log files should be monitored to find the exact issue for this bug.

With rest-fuzzer, it is possible to keep track of the code coverage, the number of executed tests, HTTP status codes, and the number of discovered vulnerabilities. The reporter generates reports with the code coverage; the fuzzer uses no coverage feedback during the fuzzing process.

Fuzzing phases The model-based blackbox fuzzer based on rest-fuzzer has two phases. The phases are the initialisation phase and the fuzzing phase.

In the first phase, initialisation, the fuzzer calculates the start- and end times for the fuzzer. Also, the fuzzer loads the potential actions on the SUT to the model. After this

phase, the fuzzer knows what actions to target on the SUT and when the fuzzer should stop.

In the second phase, fuzzing, the fuzzer gets the action from the available actions and executes that action with a value from the model. This process continues until the end time exceeds.

Code coverage Measuring code coverage is possible by running Xdebug as a debugger. Xdebug should be attached to the SUT, and the SUT should be modified to call the coverage functions. This is potentially impossible when a SUT uses an external module to handle the REST web services. In this case, it is not possible to measure the code coverage.

Xdebug gives detailed information on the included PHP files during a request and the executed lines in these files. This precision led to the decision to create the rest-fuzzer code coverage reports based on Xdebug logs.

When Xdebug monitors the code coverage, rest-fuzzer has a built-in reporter tool to convert the Xdebug log files into a LaTeX figure. This makes it possible to see the actual results easily and makes it possible to compare this fuzzer.

Responses Rest-fuzzer stores every request and response. This makes it possible to create a figure that provides a good view of what happened during the fuzzing process. Also, manually reviewing these responses to find potentially triggered vulnerabilities is possible.

3.3.4. Traditional whitebox fuzzing

For this research, the fuzzing tool PHP Vulnerability Hunter is chosen[16] as the traditional whitebox fuzzer. PHP Vulnerability Hunter is a whitebox fuzz testing tool to fuzz PHP web applications. This tool can fuzz test REST web services that have a PHP implementation. PHP Vulnerability Hunter is capable of detecting a wide range of vulnerability types based on the source code. In addition, PHP vulnerability Hunter has some built-in scan plugins that each scan for a single vulnerability like SQL injection. Besides PHP Vulnerability Hunter, no other traditional whitebox fuzzer found when searching for an open-source whitebox fuzzing tool that worked with the SUTs developed with PHP.

With a combination of static and dynamic analysis, PHP Vulnerability Hunter maps the target application based on the source code. This technique makes it possible to find entries and inputs.

As already mentioned, PHP Vulnerability Hunter can detect a wide range of vulnerability types. The list below provides an overview of the available vulnerability types:

- Arbitrary command execution
- Arbitrary file read/write/change/rename/delete
- Local file inclusion
- Arbitrary PHP execution
- SQL injection

[16] https://archive.codeplex.com/?p=phpvulnhunter

- User-controlled function invocation
- User-controlled class instantiation
- Cross-site scripting (XSS)
- Open redirect
- Full path disclosure

Fuzzing phases PHP Vulnerability Hunter executes the fuzzing process in three phases. The phases within PHP Vulnerability Hunter are the initialisation phase, the scan phase and the uninitialisation phase.

During the first phase, the initialisation phase, the source code is analysed with static analysis to discover interesting entries (function calls) and inputs. Also, backup files are initialised based on the current state of the application. Then, in the standard files, the fuzzer adds annotations used for measuring the code coverage. When an annotation action happens, this triggers a log event. After this phase, the fuzzer knows what to target and what input is necessary to fuzz the function.

The second phase, the scan phase, detects the bugs and vulnerabilities within the SUT. The fuzzer does this by scanning every file within the SUT using the integrated scan plugins. During each request, the fuzzer uses dynamic analysis to discover new inputs and vulnerabilities.

The last phase, unitialisation, is started when the scan phase is complete. Then, all original application files are restored based on the backups made in the initialisation phase. This means that the fuzzer removes all annotations from the files. Additionally, manually deleting the log file is recommended. This saves much space on the system. During the fuzzing process, log files of 1 GB are obtainable. After this phase, the application should function as it did before starting the PHP Vulnerability Hunter.

Code Coverage The fuzzer measures code coverage on the function level or the code block level. The code coverage during the fuzzing process is filtered for each scan plugin. This makes it possible to see how much of the source code is reached by each scanning plugin (e.g., a SQL injection scanning plugin). Measuring code coverage on the code block level is slower but is more accurate. The prototype uses this more accurate option to get more accurate results.

3.3.5. Model-based whitebox fuzzing

With both the model-based blackbox fuzzer and the traditional whitebox fuzzer confirmed, the used techniques can be combined to create the model-based whitebox fuzzer. In this case, the model-based whitebox fuzzer extends the model-based blackbox fuzzer with techniques from the whitebox fuzzer to gain more information about the SUT. Section 5.1 describes how the model-based whitebox fuzzing prototype works.

Finally, the model-based whitebox fuzzer is compared with the model-based blackbox fuzzer and the traditional whitebox fuzzer. This comparison is important to conclude if the hypothesis is correct and a similar model-based whitebox fuzzing approach as the approach from Pham et al. [32] works for REST web services.

4

VULNERABILITIES AND COMPARISON METRICS

This chapter covers research performed during the research period. The covered topics are the topics described in the previous chapter: vulnerabilities in REST web services, model-based whitebox fuzzing and metrics to compare fuzzing methods. This chapter answers research question 1 and provides extra information for research question 3.

4.1. VULNERABILITIES

It is necessary to consider areas where an attacker could exploit a vulnerability to secure a REST web service. The Open Web Application Security Project (OWASP) Foundation provides best practices for making a REST web service secure. The remaining part of this section discusses vulnerabilities affecting REST web services. In addition, each subsection notes if a vulnerability is fuzzable or non-fuzzable. The sources used for this overview are OWASP API Security Top 10 Cheat Sheet [4], OWASP REST Security Cheat Sheet[27], Postman Top 5 API Security Best Practices for 2021[19].

4.1.1. HTTPS

Secure REST web services should provide only HTTPS endpoints. When unsecured HTTP endpoints are used, the authentication credentials, e.g., passwords and API keys, are not protected during transit. Also, integrity can not be guaranteed, and the service is not authenticated because the client does not allow it.

This vulnerability is detectable in the OpenAPI specification of the SUT. Within the specification, not all servers will have an URL that starts with https://. In those cases, the fuzzer should mark it as a vulnerability.

This vulnerability is fuzzable for the model-based fuzzers. Both model-based fuzzers will use the OpenAPI specification. In this specification, it is documented if HTTP or HTTPS is used. For the traditional whitebox fuzzer, this data is not available during the execution of the fuzzer. The traditional whitebox fuzzer does not have a scanning method for this vulnerability. This is also not fuzzable when the fuzzer is done; the executed requests do not specify the HTTP method (HTTP or HTTPS).

4.1.2. Access Control

Access control at each REST endpoint is a must for non-public REST web services. This is necessary to avoid issues with the unauthorised use of REST endpoints that perform changes on the database. OWASP suggests that each endpoint should be self-responsible for access control to minimise latency, reduce coupling between services, and a centralised Identity Provider should issue the access tokens [27].

Public REST web services without access control run the risk of being farmed, leading to excessive bills for bandwidth or compute cycles. User authentication mitigates this risk [4]. User authentication, including API keys, is centralised in an Identity Provider (IdP), the IdP issues access tokens [27]. With API keys, access can be given based on a valid API key. This caller places the API key within the request header. Before the REST web service processes the request, a check is done if the API key is valid or revoked. In addition, API keys can reduce the impact of denial-of-service attacks since revoked keys will not start a high-performance operation on the REST web service. Furthermore, based on the API key, the HTTP status code 429, "Too Many Requests", can be returned when a client performs more than accepted requests within a given time.

Besides API keys, web services could also use JSON Web Tokens (JWT) for authentication. JWTs are JSON data structures containing a set of claims that act as a signature [27]. The message authentication code (MAC) could be used to protect the integrity of the JWT.

One of the well-known consequences of broken access control is declined access, and access privilege alteration, one of the most common attacks by attackers[19].

A REST web service endpoint should return a 401 "Unauthorised" HTTP status code when the authentication token is missing. A request without an authentication token in the header that leads to a 2XX status code has an access control vulnerability.

This vulnerability is fuzzable for the model-based fuzzers. It is possible to see the SUT response when an API key is included in the request. However, there is nothing implemented for the traditional whitebox fuzzer that makes it possible to analyse the access control vulnerabilities.

4.1.3. Restrict HTTP methods

When the client can request an unauthorised resource, unexpected behaviour could happen, such as inserting data in the database or, worse, deleting data in the database.

OWASP suggests creating a list of permitted HTTP methods, e.g. CREATE, GET, PUT, and DELETE[4]. HTTP methods not listed should be rejected with HTTP Status Code 405; Method not allowed. In addition, the client requesting the HTTP method should be authorised to use the HTTP method on the database.

When an HTTP method on an endpoint is not specified, the endpoint should return a 405 HTTP status code. When this is not the case, then there is a vulnerability that a client can perform a request to an unauthorised resource. The model-based fuzzers are using the OpenAPI specification to create their models. Based on this model, it is impossible to call an HTTP method not included within the model. The traditional whitebox fuzzer generates the requests based on the source code; by default, no not specified HTTP methods are used during this method.

4.1.4. Input validation/Injection

Input validation issues are probably the most common issues found during the fuzzing process. When not enough validations are implemented, an invalid input can cause crashes within the REST web service or update/delete resources. These issues are linked to injection attacks, e.g., SQL, command and NoSQL [4]. Additionally, cross-site scripting (XSS) is an attack resulting when the input validation is insufficient[19]. OWASP created the following rules to deal with input validation[27]. All rules limit the chance of a sound looking invalid input.

- Do not trust the input parameters/objects.
- Validate input: length / range / format and type.
- Achieve an implicit input validation using strong types like numbers, booleans, dates, times or fixed data ranges in API parameters.
- Constrain string inputs with regular expressions.
- Reject unexpected/illegal content.
- Use validation/sanitation libraries or frameworks in the specific language.
- Define an appropriate request size limit and reject requests exceeding the limit with HTTP status code 413 Request Entity Too Large.
- Consider logging input validation failures. Assume that someone who is performing hundreds of failed input validations per second is up to no good.
- Use a secure parser for parsing the incoming messages. If XML is used, use a parser that is not vulnerable to XXE (XML External Entity) and similar attacks.

Different types of injection attacks can be performed. The fuzzer can detect this vulnerability by checking if an injection attack succeeds. Regular expressions should be able to check if the request contains an injection attack. When a following dependent GET request is performed, the response should contain an unharmful version of the injection input. When this is not the case, an injection attack succeeded on the endpoint. Checking the responses for injections is possible. This makes this vulnerability fuzzable for all fuzzer types.

4.1.5. Validate content types

The body of a REST request and the body of a REST response should match the intended content type in the header. Otherwise, this could cause misinterpretation at the client or server-side and lead to code injection/execution[27].

Documenting all supported content types in the API specification is the solution for this issue. In this case, requests with an unsupported content type or missing content type are rejected and responded with the HTTP Status Codes 406 "Unacceptable" and 415 "Unsupported Media Type".

Fuzzing with unexpected content types could potentially lead to some unexpected behaviour. REST web services most likely support various content types, e.g., application/json and application/xml. The response should contain a 406 or 415 HTTP status code

when the request body or response does not match the header's content type. If this is not the case, an invalid content-type vulnerability could occur. Analysing the requests and responses makes this vulnerability fuzzable for model-based fuzzers. The traditional white-box fuzzer does not have the analysing tools to find this vulnerability.

4.1.6. MANAGEMENT ENDPOINTS

Management endpoints within REST web services are implemented endpoints not intended for regular use. Their use is maintaining the application, and the users with access should only call these endpoints. OWASP made some recommendations for the security of management endpoints[27].

- Avoid exposing management endpoints via the Internet.
- If management endpoints must be accessible via the Internet, users must use a strong authentication mechanism, e.g. multi-factor.
- Expose management endpoints via different HTTP ports or hosts, preferably on a different network interface card and restricted subnet.
- Restrict access to these endpoints by firewall rules or use of access control lists.

The API specification does not specify which endpoints are management endpoints. This makes it not possible to find these endpoints and test them. The traditional whitebox fuzzer can find the management endpoints. However, based on the source code, it is impossible to define the endpoints as management endpoints. This vulnerability will not be considered for the model-based whitebox fuzzer and model-based blackbox fuzzer as it is non-fuzzable. In those cases, having a specification would be helpful.

4.1.7. ERROR HANDLING

Sometimes, it is easier to program with good feedback. In those cases, it can be helpful to return some details about what stack trace provided the crash. An attacker could use this information to generate more targeted attacks. A best practice is not to reveal technical information (e.g., call stacks or internal hints) of the failure unnecessarily to the client. For REST web services, responses could reveal detailed information. Here the caller must know what went wrong without exposing too much information, e.g., the connection strings [12] to the database.

Good feedback, e.g., stack traces or internal hints, could help attackers. When an error occurs, the response should not contain this information. This vulnerability is fuzzable by checking the responses when an HTTP 5XX status code is returned. Regular expressions could be used to search responses for detailed information. All fuzzers have the option to scan the responses for error handling vulnerabilities.

4.1.8. AUDIT LOGS

REST web services use audit logs to store auditable events, such as logins, failed logins, and high-value transactions [5]. Audit logs could contain sensitive information. A log injection attack can provide an attacker with this information. In addition, an attacker could cover his tracks by removing these audit logs. OWASP delivers some recommendations to make the audit logs less sensitive [27].

- Write audit logs before and after security-related events.
- Consider logging token validation errors to detect attacks.
- Take care of log injection attacks by sanitising log data beforehand.

The audit logs can be corrupted when an attacker is trying to add audit logs within its requests. The fuzzer is not able to detect this vulnerability when no audit logs are used. For the fuzzers, this vulnerability is marked as non-fuzzable. None of the fuzzers performs analyses on the audit logs.

4.1.9. Sensitive information in HTTP requests

REST web services should prevent leaking credentials. Passwords, security tokens, and API keys should not be visible in the URL [27]. The browser can store these requests, making them readable for attackers [19].

To prevent sensitive information in HTTP requests, POST and PUT requests should transfer sensitive data only in the request body or header. In this case, it is not stored directly. In a GET request, the URL should not add any critical information to the path. The URL https://api.example.com/resource/123 is ok since it does not include sensitive data. However, the URL https://api.example.com/resource/123?apiKey=abcdefghijklm is not ok, since it consists of the API key.

The model should be checked to see if sensitive information is included in the path parameters. With regular expressions, common sensitive words should be searched. Potential words like key, password, and credit cards could contain sensitive information and should not be included in the URL. This vulnerability is fuzzable by checking if the request contains sensitive words in the path. The model-based fuzzers allow the user to analyse this. By checking the input parameters, a static analysis could make it possible for the traditional whitebox fuzzer. However, this check is not implemented on the traditional whitebox fuzzer.

4.1.10. Excessive data exposure

The REST web service could be exposing more data than is needed by the client. In this case, the client is responsible for filtering the data. However, relying on the client is a vulnerability since this data could contain sensitive data which the client does not need[4].

The same applies to error handling and audit logs described prior. Preventing this vulnerability can be done by not letting the client filter the data. Furthermore, response checks could be enforced to avoid accidental leaks of data.

This vulnerability is hard to detect without knowing the purpose of the response. If sensitive information is not used, it should not be included. However, to define it as excessive data exposure, more information is necessary. Based on a path and input and output parameters, it is impossible to say if the expected output contains too much data. For this reason, this vulnerability will not be taken into account for the model-based fuzzers (non-fuzzable). The traditional whitebox fuzzer contains no analysing functions for this vulnerability.

4.1.11. BROKEN FUNCTION-LEVEL AUTHORISATION

Attackers can replace the function name with another function name. This could trigger a hidden API method that was potentially not documented. When no access control is implemented, the attacker could obtain admin functions or admin data [4]. REST web services most often call a different function when the path is modified. For example, when a request tries to access the path `GET /api/comments`, a function tries to get the comments. When an attacker replaces "comments" with "passwords" (`GET /api/passwords`), the attacker tries to trigger a function that returns all passwords.

Denying access from user accounts to admin functions or restricting access by default could prevent this vulnerability from happening.

The model-based whitebox fuzzer will use a model to generate tests. This makes it impossible for the fuzzer to detect this vulnerability because no function is targeted, which is not documented within the specification. This vulnerability will not be considered for the model-based whitebox fuzzer and model-based blackbox fuzzer (non-fuzzable). The traditional fuzzer performs scans on the code and could find these functions. However, the traditional whitebox fuzzer does not know which function is hidden.

4.1.12. LACK OF RESOURCES AND RATE-LIMITING

The REST web service is not protected against a large, excessive number of calls or payload sizes. This vulnerability could lead to a Denial of Service (DoS) attack and other brute force attacks, such as crack passwords[4] and Distributed Denial of Service (DDoS)[19].

Limiting payload sizes and defining proper request rates are two countermeasures to prevent this vulnerability.

The HTTP status code 429, "Too Many Requests", is used to slow down clients from sending large, excessive numbers of calls or payload sizes. If the REST web service never responds with this HTTP status code, there are two options: the limit is not reached yet, or the endpoint is vulnerable to this attack. This threshold should be set manually but is not included in the fuzzer. This makes this vulnerability non-fuzzable during this research. Additionally, the environments will not have a rate limit making this vulnerability non-fuzzable for the fuzzers. This choice makes it possible to have more requests within the given time.

4.1.13. SECURITY MISCONFIGURATION

Poor configuration of the REST web services could allow attackers to exploit them[4]. When systems are unpatched, files and directories are unprotected, and unnecessary features are enabled. Various steps can be taken to improve the security configuration to prevent these vulnerabilities.

This vulnerability includes inadequate/not protected endpoints. This could happen when, for example, CORS[1] is enabled. The OpenAPI specification can be used to check if CORS is enabled. However, it is only called a misconfiguration when it was not designed to support this function. For a fuzzer, it is not possible to know if it was an accidental misconfiguration of a design choice. This is why the vulnerability will not be taken into account for the model-based whitebox fuzzer. Furthermore, this vulnerability is also not supported by the model-based blackbox fuzzer and traditional whitebox fuzzer.

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

4.1.14. IMPROPER ASSETS MANAGEMENT

Attackers can sometimes find non-production versions of REST web services (for example, beta, acceptance, and testing) that are not as well protected as the production version. As a result, attackers could use these non-production versions to gain information and create attacks[4].

Removing these non-production versions or adding proper security on these non-production versions is the best option to counter its vulnerability.

This vulnerability can occur when a non-production version is available that could leak information. In this case, this vulnerability will not be considered in this research because the location of OpenAPI specification is configurable (non-fuzzable). The model-based blackbox fuzzer and the traditional whitebox fuzzer do not fuzz this vulnerability. For the model-based fuzzers, it would be more accessible to fuzz eventually; when an older version of the OpenAPI specification of the API specification is available, this specification could leak the endpoints hidden in the recent versions.

4.1.15. INSUFFICIENT LOGGING AND MONITORING

When there is a lack of proper logging, monitoring, and alerting, attacks and attackers go unnoticed[4]. Setting up adequate logging, monitoring, and warning is the only option to counter this vulnerability. For REST web services, monitoring users' activity helps to discover users trying to exploit the system. If the system logs the attacker's information/API key, the system could block the attacker from accessing resources by revoking the API key.

This vulnerability occurs when there is a lack of logging. Source code should be analysed to find if the endpoint logs the request. However, the model does not include this information. Due to this, this vulnerability will not be considered for the model-based whitebox fuzzer. This vulnerability is also non-fuzzable for the model-based blackbox fuzzer for the same reason. The traditional whitebox fuzzer is the only fuzzer that scans the source code. However, the chosen traditional whitebox fuzzer does not scan the source code for this vulnerability.

4.1.16. SUMMARY

This summary contains a table showing the results of all vulnerability types. In addition, for each vulnerability type, it is noted per type of fuzzer whether it is fuzzable.

Table 4.1: Vulnerability types per fuzzer type

Vulnerability type	**Model-based whitebox fuzzing**	**Model-based blackbox fuzzing**	**Traditional whitebox fuzzing**
HTTPS	✓	✓	-
Access control	✓	✓	-
Restrict HTTP methods	-	-	-
Input validation/injection	✓	✓	✓
Validate content types	✓	✓	-
Management endpoints	-	-	-
Error handling	✓	✓	✓
Audit logs	-	-	-
Sensitive information in HTTP requests	✓	✓	-

Continued on next page

Table 4.1 – continued from previous page

Vulnerability type	Model-based whitebox fuzzing	Model-based blackbox fuzzing	Traditional whitebox fuzzing
Excessive data exposure	-	-	-
Broken function-level authentication	-	-	-
Lack of resources and rate-limiting	-	-	-
Security misconfiguration	-	-	-
Improper assets management	-	-	-
Insufficient logging and monitoring	-	-	-

4.2. METRICS

This research uses metrics to compare the different fuzzing methods. This part of the study focuses on identifying valuable metrics. A performed literature study obtains information about the metrics.

Nowadays, comparing fuzzing methods happens based on several metrics: the number of discovered vulnerabilities [32], the number of executed tests [35], code coverage [7], and the HTTP status codes [7]. The subsections below describe the different metrics.

4.2.1. NUMBER OF DISCOVERED VULNERABILITIES

The number of discovered vulnerabilities indicates how many vulnerabilities are found by the fuzzer. This metric is useful when comparing different fuzzers with each other. When vulnerabilities are found, it is easy to see which fuzzer could find them.

The problem with this issue is that fuzzing is a random process. This means that each run will have different inputs, which will potentially lead to different results. This makes it hard to derive conclusions based on this stand-alone metric. Also, when the fuzzer finds no errors, this metric does not provide any information about the quality of the input data generated or mutated [35].

4.2.2. NUMBER OF EXECUTED TESTS

The number of executed tests indicates how effective a fuzzer is. When the fuzzer cannot generate inputs that the SUT accepts, the fuzzer will eventually execute more tests. Good inputs will take longer to process since they enter the SUT on a deeper level or are less frequently rejected in an early stage.

With the number of executed tests, the executor of the tests can measure the effectiveness of a fuzzer. However, stand-alone, it is hard to derive a conclusion. The number of executed tests can be influenced by the SUT or fuzzing method. Combining this metric with other metrics could provide a more precise conclusion.

4.2.3. CODE COVERAGE

Code coverage indicates the percentage of the source code that is executed by the fuzzer, assuming that higher code coverage is a result of a more effective fuzzer. The code coverage is measured by calculating the total number of instructions, basic blocks, and routines executed in the SUT [35].

However, code coverage does not consider the complexity of the executed code coverage of the SUT. This means that different functions can have the same number of lines of code, but the complexity of different functions is not the same. Generally, increasing the complexity leads to an increased chance of vulnerabilities[37].

4.2.4. HTTP STATUS CODES

When looking at the HTTP status codes, the number of valid requests can be determined. For web services, there are five classes defined:

- **1XX informational**
  The request has been received, and the process is continuing.
- **2XX success**
  The request was successfully received, understood, and accepted by the server.
- **3XX redirection**
  Further action(s) needs to be taken to complete the request.
- **4XX client error**
  The request contains bad syntax or cannot be fulfilled.
- **5XX server error**
  The server could not fulfil a (valid) request.

From these five classes, the 4XX and 5XX classes represent errors. When a response contains a 4XX or a 5XX status code, the conclusion can be made that the request was not valid for some reason. For example, when the response contains a 404 status code, the server could not find the requested resource. The server errors are linked to the 5XX errors. For example, when a 500 status code is found, the server caught an unexpected condition during the execution that prevents the server from fulfilling the request. A 501 HTTP status code appears when a REST service is not yet implemented. A 503 HTTP status code is returned when the service is (temporarily) unavailable for processing the incoming request.

For the other three (1XX, 2XX, and 3XX) classes, when the response contains one of those status code types, the conclusion can be made that the request was received and is valid. The most common status codes are 200 (OK), 201 (Created), 202 (Accepted), and 204 (No content). These status codes will most likely occur when using the POST, GET, PUT, and DELETE HTTP methods.

Measuring the HTTP status codes from responses from REST web services is possible. Each response contains an HTTP status code. If the fuzzer monitors the HTTP status codes, this can say something about the ability of a fuzzer to create valid inputs. Combining the HTTP status codes with the number of executed tests can provide more insight into how effective a fuzzer generates valid inputs. However, this mainly involves the input generation strategy. As mentioned, a generation-based approach most likely generates more valid inputs than a mutation-based approach.

4.2.5. CONCLUSION

When looking at the previous subsections, the conclusion can be made that none of the described metrics would provide enough information to create a conclusion. The best solution would be to combine the different metrics.

The number of discovered vulnerabilities, the number of executed tests and the HTTP status codes provide an overview containing useful information. From this information, it can be concluded how many tests were run, how many vulnerabilities were found, and how effective was the fuzzer with generation inputs. Combining this with code coverage, which gives insight into how many lines of code are executed, can lead to a set of metrics that can be used.

Eventually, adding the total complexity and the executed complexity of the tested SUT will provide a total overview.

- **Number of discovered vulnerabilities** - How many vulnerabilities were discovered within the SUT?
- **The number of executed tests** - How many tests were executed within the time limit?
- **Code coverage** - Which percentage of the SUT is tested?
- **Status code categorisation** - How are the executed inputs processed, and what status codes did they return?

5

DEVELOPING A MODEL-BASED WHITEBOX FUZZER PROTOTYPE

With the performed studies in the previous chapters about the usage, specifics, limitations, vulnerabilities of REST web services, and the prior research, a design for the model-based whitebox fuzzer can be created. The main goal of this fuzzer is to validate the model-based whitebox fuzzing technique. This fuzzer will gather the necessary inputs for the required parameters of a tested endpoint. The following sections will contain the design for a model-based whitebox fuzzer.

Firstly, section 5.1 describes the algorithm for the model-based whitebox fuzzer. Secondly, section 5.2 describes the architecture of the prototype. Thirdly, section 5.3 covers the implementation of the prototype. Fourthly, section 5.4 includes the validation of the prototype. Finally, section 5.5 consists of the results from running the different fuzzing techniques.

5.1. MODEL-BASED WHITEBOX FUZZING

The developed prototype will follow the algorithm described in listing 5.1. This listing contains the model-based whitebox fuzzing algorithm from Pham et al.[32], see listing 2.1, with some modifications to make it fit for REST web services.

5.1.1. FUZZING PHASES

The model-based whitebox fuzzer has two phases during the execution of the fuzzer. Then, when the fuzzer completes, a third phase is available to convert the results into figures.

The fuzzer starts in the preparation phase. This phase is responsible for identifying the available target locations (endpoints) and instantiating valid inputs, outputs, and dependencies based on the OpenAPI specification. The fuzzer places this data within a semi-formal model. When this completes, the fuzzer will have a complete model containing the necessary information for the next phase.

When the preparation is complete, the fuzzer enters the fuzzing phase. During this phase, the fuzzer will continuously loop through a sequence of actions. The sequence starts by selecting the target location within the SUT. Based on the selected target location, the fuzzer generates a valid input file. Next, the fuzzer performs some manipulations on the input to fuzz different values. When the file is ready, the fuzzer sends it to the SUT. The SUT

will process the input and generate a response. The fuzzer will monitor the execution of the test on the SUT, and the obtained information is stored. The fuzzer stores the code coverage details locally within a log file, and the response from the SUT is stored in a database. By storing the response in the database, the model can include this response for new test cases.

When the fuzzing phase exceeds the timeout, the fuzzer will finish its last request and end the process. The results are stored within the database and are available for analysis at that point. The fuzzer has a third phase, reporting, making it possible to create helpful figures based on the database values.

5.1.2. ALGORITHM

The developed algorithm covers the second phase of the fuzzing process, the fuzzing phase. The algorithm is a modified version of the original model-based whitebox fuzzing approach in listing 2.1. The following description provides a detailed description of algorithm 5.1. Also, this subsection describes the modifications to the original algorithm.

The algorithm takes a Program P, an input Model M, test execution services T, and an empty set of target locations L. From these inputs, the Program P is the SUT. The fuzzer obtains the input Model M, uninstantiated test execution services T and the empty set of target locations L after the preparation phase, when the fuzzer obtains the information from the REST web service (OpenAPI) specification. The target locations L and information for the test execution services T come from the input Model M. This can be seen in lines 1-7 of listing 5.1. The algorithm obtains the target locations from the model of the SUT, including a list of dependencies on line 2. On line 5, the fuzzer initiates the test execution services. The fuzzer initiates the test execution services by initiating the required services for the fuzzing process. For example, the fuzzer obtains the dependencies between the endpoints of the model. In addition, the fuzzer could set the authentication information if provided.

Listing 5.1: Algorithm model-based whitebox fuzzing for REST web services

```
Input: Program P, Input Model M
Input: Test execution services T, Targets L
Output: Test results T'
1: if L = ∅ then
2:         L ← IDENTIFYCRITICALLOCATIONS(P)
3: end if
4: if T = ∅ then
5:         t ← INSTANTIATEASVALIDINPUT(M)
6:         T ← {t}
7: end if
8: while timeout not exceeded do
9:         Target location l ← CHOOSETARGET(L)
10:        Input file t ← CHOOSEBEST(M)
11:        Fragment Pool Φ ← FILECRACKER(T,M)
12:        Valid file f ← FILESTITCHER(t, Φ,M)
13:        Valid file f' ← FILEREPAIR(f,M)
14:        T' ← T ∪ f'
15: end while
```

The main loop of the model-based whitebox fuzzing algorithm is shown in lines 8-15. While the timeout is not exceeded, model-based whitebox fuzzing chooses the next target location l on line 9. Based on the model, the fuzzer selects a dependent endpoint of the previously tested endpoint. A dependent endpoint is an endpoint that required data from the original endpoint or an endpoint that accesses the same resource; more information is available in section 5.2.3. Otherwise, when there are no more dependent endpoints, the fuzzer selects the following endpoint from the set of available locations. It is important to make the fuzzer stateful. This is important because a stateful fuzzer can choose better locations based on the data from previous requests. With these better locations, the chances of valid requests are higher, meaning that the fuzzer could potentially reach deeper within the SUT [32]. For this reason, the fuzzer creates sequences of dependent endpoints. The fuzzer selects the target location l from these sequences of dependent endpoints.

On line 10, the fuzzer selects the input file t by looking at model M. This part is modified because the best input file can be chosen based on the model while fuzzing REST web services. This model contains the expected format for the endpoint. Model M also contains the outputs from the previous requests, which are available for future requests. The file cracker uses these files to construct a fragmented pool Φ on line 11. The fuzzer uses the fragmented pool to get values for parameters for dependent endpoints.

The algorithm uses the file stitcher on line 12 to create a file that contains valid non-fuzzable parameters. The fuzzer generates these non-fuzzable parameters based on the fragmented pool Φ; this fragmented pool is empty at first but gets filled during the execution. The fuzzer generates other fuzzable parameters randomly based on their type.

After modifying the input l, the fuzzer uses file repair to build the input on line 13. When the fuzzer creates a valid input, this input acts as a new test case. On line 14, the fuzzer inserts the test case in the test execution services T. The test execution services T execute the test case on the SUT and store the response T'.

5.2. Architecture

The developed prototype is a Java web application. The used architecture for the fuzzer builds on the architecture used for a model-based fuzzer, also known as rest-fuzzer[15]. This fuzzer has been proven efficient for model-based fuzzing when comparing it with blackbox fuzzing and dictionary fuzzing. Therefore, the decision to build the model-based whitebox fuzzer on top of rest-fuzzer is based on the results from that research. However, the fuzzing process will have a different implementation to function as a model-based whitebox fuzzer.

5.2.1. High-Level Design

The architecture of the prototype consists of two applications, the back-end and the front-end. The back-end application is responsible for running the entire fuzzing process. The front-end application is a GUI for users of this prototype. The browser visualises the data and results for the user.

The three-layered architecture is the base for the architecture of the model-based whitebox fuzzer[15]. Two applications implement these three layers. The front-end application realises the presentation layer, and the back-end application realises the domain and data source layers. Figure 5.1 presents an overview of the modules within the different layers.

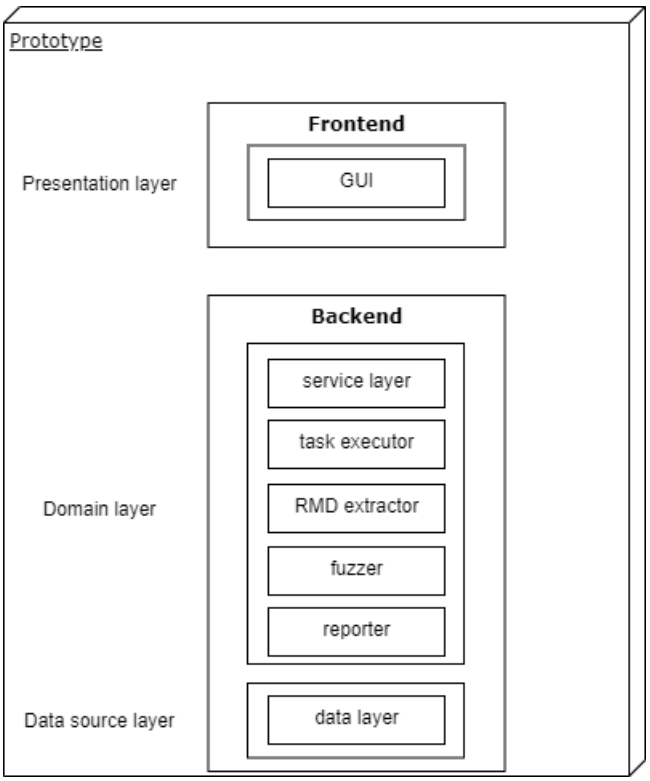

Figure 5.1: An overview of the modules within the three layers from the prototype.

Within the domain layer, the back-end application implements the service layer. This service layer controls the transactions and the security. The total domain layer consists of the following modules: the service layer, the task executor, the REST model description (RMD) extractor, the fuzzer, and the reporter.

The data source layer consists of a data layer. This layer is responsible for the communication between the application and the database. Hibernate [1] is a framework that manages database activities. This results in that the data layer only contains models of the classes necessary to communicate with the database and the application.

During the fuzzing process, the fuzzer will walk through multiple phases. In the preparation phase, the main goal of the fuzzer is to obtain the necessary information about the REST web service and create a model. This is done by parsing the specification of the REST web service. Section 5.2.2 describes how this process works. In this preparation phase, the fuzzer infers the dependencies of the different endpoints in the REST web service. Inferring dependencies is described in section 5.2.3. Combining the obtained information leads to the model. The fuzzer can use this model for later input generation. Within the preparation phase, the fuzzer makes no requests to an endpoint but obtains information to set up the test execution services. These steps complete lines 1-7 from the fuzzing algorithm described in listing 5.1.

Within the next phase, the fuzzer will actually start creating and executing test cases. This main phase is divided into several subphases. These subphases are scheduling, input generation, configuration updating, and evaluation. In the scheduling phase, the fuzzer determines the following target location in the SUT. The input generation phase generates the input for the test case. The fuzzer is a generation-based fuzzer based on the model

[1] https://hibernate.org/

inferred from the preparation phase. Model-based generation creates an input schema for the endpoints.

After the execution of a test case, the evaluation and configuration updating phases start. In the configuration updating phase, the model inferred from the preparation phase is updated based on the outcome of the executed test case. Doing this provides a better model for input generation because the newly obtained values are available as parameters. Next, the evaluation phase checks the executed test and looks if the request returns an HTTP 200 status code.

Figure 5.2 displays a simplified overview of the proposed process for the fuzzer module.

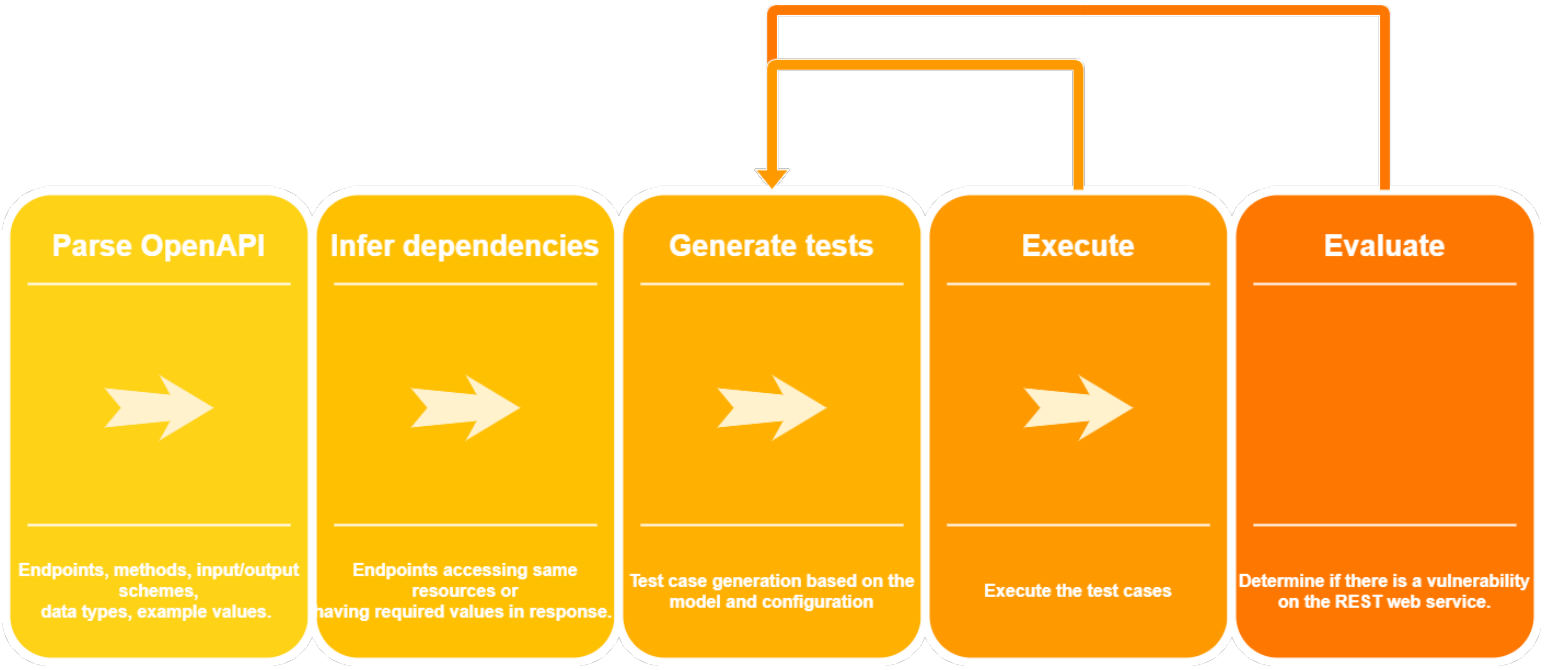

Figure 5.2: Proposed fuzzer module process

5.2.2. Parsing OpenAPI

The fuzzer parses the OpenAPI specification to create a model. The used model is a semi-formal model because it makes it possible to allow structured, automated testing of complex applications. A formal model would also be possible, but setting up the formal model would take much time for complex applications because all paths must be included to prove the properties mathematically. An informal model would make it difficult to model the complexity of these complex applications [30]. Due to this, the semi-formal is the most suitable option. The semi-formal model consists of the interface description, the behavioural model and the deployment information to allow structured, automated testing [20].

First, the model-based whitebox fuzzer obtains the interface description and the deployment information of the model from the SUT for the input generation. In the model of a REST web service, the interface description contains a list of the available endpoints, input parameters and outputs for the endpoints. The deployment information consists of the host name. The fuzzer derives both parts of the model from the OpenAPI specification rather than from code [27]. The OpenAPI specification of REST web services contains

information that is easy to understand for machines and humans. In addition, the implementations for endpoints are relatively short and access different systems.

As mentioned previously, the OpenAPI specification is one of the used techniques used to define the specification. Within this research, the fuzzer uses the OpenAPI specification to generate inputs. Listing 5.2 provides an example of the OpenAPI specification from the SutSqlI project. This OpenAPI specification describes the web services for SutSqlI. The listing describes a single path from the OpenAPI specification; in this case, "rest/comments". The OpenAPI specification describes each HTTP method on a path and contains the input parameters and potential return values.

Listing 5.2: OpenAPI Specification SutSqlI

```
"paths": {
    "/rest/comments": {
        "get": {
                "tags": [
                        "comment-controller"
                ],
                "summary": "read",
                "operationId": "readUsingGET",
                "produces": [
                        "*/*"
                ],
                "parameters": [
                        {
                                "name": "id",
                                "in": "path",
                                "description": "id",
                                "required": true,
                                "type": "integer",
                                "format": "int64"
                        }
                ],
                "responses": {
                        "200": {
                                "description": "OK",
                                "schema": {
                                        "type": "object"
                                }
                        },
                        "401": {
                                "description": "Unauthorized"
                        },
                        "403": {
                                "description": "Forbidden"
                        },
                        "404": {
                                "description": "Not Found"
```

```
                        }
                    }
                },
            },...
}
```

From this OpenAPI specification, the fuzzer obtains the necessary information for creating a model. The model will consist of specifications for each endpoint. The remainder of this subsection describes the process of obtaining information from the specification.

The OpenAPI specification makes it simple, understandable and straightforward to process the specification of an endpoint. Firstly, the endpoints should be saved. Each endpoint with multiple HTTP methods will result in an endpoint for each HTTP method. So based on listing 5.2, `GET /rest/comments/{id}` and `DELETE /rest/comments/{id}` will result in two different endpoints. For each endpoint, the model stores the following values:

- The base URL for the endpoint
- The input schema for path parameter(s)
- The input schema for the request body
- The input schema for form data
- The input schema for query parameters
- The responses schema
- The list of parameters in the response
- The required property for the parameters

It is necessary to store these values for each endpoint for the next part of the fuzzing process.

5.2.3. Inferring dependencies

The second step is inferring dependencies from the OpenAPI specification. By defining the dependencies of an endpoint, the behavioural model is created. The first part of this step is determining what should be considered a dependency of an endpoint.

A dependency is a value for a parameter required by the endpoint's input schema. With this in mind, for the REST web service, a dependency is data needed to make an API call. Therefore, dependencies for an endpoint are obtainable from the responses of other endpoints described by the OpenAPI specification. For example, based on the OpenAPI specification in listing 5.2, `GET /rest/comments/{id}` requires an id. In this case, a dependent endpoint would be an endpoint that returns a response containing a property with the name "id".

Besides the dependencies gained from the responses of endpoints, there is another method to gain dependencies. Other endpoints accessing the same resource are dependencies of an endpoint as well. For example, based on the OpenAPI specification in listing 5.2, a new endpoint `DELETE /rest/comments/{id}` would also access the same resource as `GET /rest/comments/{id}`. This would make those endpoints dependent on

each other.

Together the following list shows the dependent endpoints of a target endpoint:

- Endpoints using at least one of the required parameters for the target endpoint in their response schema.
- Endpoints having the same required parameters as the target endpoint.

Listing 5.2 displays an example for `GET /rest/comments/{id}` to give more insight on the dependencies. By looking at the required parameters from the schema, the schema only shows one required parameter named "id". None of the endpoints is returning a parameter named id. Based on the first point, there is no dependency for this endpoint. However, multiple other endpoints are accessing this resource. `PUT /rest/comments/{id}` and `DELETE /rest/comments/{id}` access the same resource. Combining both results provides a list of dependent endpoints. The list of dependent endpoints for `GET /rest/comments/{id}` consists of `PUT /rest/comments/{id}` and `DELETE /rest/comments/{id}`.

5.2.4. Test Case Creation

Test case creation consists of multiple phases, including scheduling, input generation, and configuration updates. During the scheduling phase, the fuzzer selects the configuration. The input generation phase generates the inputs for the scheduled tests based on the JSON values from the OpenAPI specification. After the execution of the test case, the fuzzer updates the configurations for the next test case. This part will cover lines 8-15 of the fuzzing algorithm in Listing 5.1.

Scheduling Based on the configuration information available for model-based whitebox fuzzing the fuzzer chooses the following target location. The original model-based whitebox fuzzing approach uses the initial seed inputs T to generate an input for the critical location l to generate inputs that could expose a vulnerability [32]. The critical locations have to be identified, and a model-based search has to be applied to do this.

The proposed model-based whitebox fuzzing approach identifies the critical locations by looking at the available endpoints. A critical location is an endpoint that may contain or expose a vulnerability when exercised by an appropriate input. Critical locations are defined by looking at locations with many dependencies.

Traditionally, model-based whitebox fuzzing uses multiple techniques to perform the model-based search. Furthermore, model-based whitebox fuzzing uses a model-based search to reduce the distance to l (target location). Since REST web services mainly consist of single-function endpoints, it is most important to test the dependent endpoints. However, for REST web services, looking at the dependencies can be enough. Endpoints that are close to the same resource can be selected fast based on the OpenAPI specification mentioned in section 5.2.2.

Input Generation The input generator will generate input just before the execution of the test case. Doing this enables the input generator to generate an input based on the results

and feedback from the previous test runs. The generated input consists of two parts: the input schema and the input parameters.

Firstly, the schema of the desired input is generated based on the target location. This is necessary to make the requests more dynamic. The fuzzer creates an input that consists of a subset of all possible parameters. The previous outputs in the model are used as donor files to create a fragmented pool for data transplantation. In the original model-based whitebox fuzzing approach, this process is called file cracking[32]. For example, `GET /rest/comments/{id}` has a query parameter id; this only returns the comment with that matching id. Next, the fuzzer could test all possible combinations of these parameters. Three types of input parameters are available during this process:

- **Required fuzzable parameters** are parameters defined as required in the OpenAPI specification. When the input does not include this parameter, the API will reject the API request from the client/fuzzer. The fuzzer generates this value based on the pool of received values. The fuzzer generates these parameters randomly.
- **Required non-fuzzable parameters** are parameters defined as required in the OpenAPI specification. A parameter is defined as non-fuzzable when a generated value will most likely result in a 404 status code. An example is the id parameter of the `GET /rest/comments/{id}` of the SutSqlI testing application. Without obtaining the id from dependencies, the request most likely is rejected by the service.
- **Optional fuzzable parameters** are parameters defined as optional in the OpenAPI specification. The fuzzer always generates the values assigned to these parameters randomly, and these parameters are not required in an API request to the REST web service.

Secondly, the input values need to be generated based on input data types from the model. The fuzzer generates the inputs randomly based on the expected data type in the model. In this step, the fuzzer uses data from the dependencies. If the parameter is marked as required non-fuzzable, then a value from the responses of dependent endpoints is used when available. In the original model-based whitebox fuzzing approach, this process is called file stitching[32] since the input data is generated based on specifications from the model.

While generating the input schema, there is a difference between the required fuzzable parameters and the required non-fuzzable parameters. Based only on the required attribute in the OpenAPI specification, these groups can not be defined. This leaves two options, manually adding this parameter to the specification for each endpoint or automating this process. However, the fuzzer is not aware of what is non-fuzzable. Therefore, the fuzzer should mark parameters as non-fuzzable during run-time. By monitoring the 404 status codes in the responses, a score property can keep track of this. If a parameter results in too many 404 status codes, the fuzzer should mark this parameter as non-fuzzable.

Finally, the fuzzer should reestablish the integrity of the file. This is a task for a file repair tool[32]. The file repair tool validates that the generated input data is a valid object.

Configuration Updating When a test case completes, the fuzzer should monitor the results to update the model for a better input generation for the following test cases. In addition, the following data should be monitored for each request: the state, the status code,

values from the output used as inputs for required non-fuzzable parameters for dependent endpoints.

The fuzzer uses the monitored data to update the model. If a test case is a follow-up test case for the previous test, or when the parameter is required non-fuzzable, the input generator will use the value from the dependencies instead of randomly generating a value with a chance of being rejected by the endpoint. As already mentioned in section 5.2.4, the fuzzer only marks a parameter as required non-fuzzable when it results in too many 404 status codes. However, the same parameter will result in the expected 2XX status codes and could also be marked as required fuzzable. To determine if a parameter is required non-fuzzable or required fuzzable, a score keeps track of this. This score is a guideline for the fuzzer when the percentage of 404 status codes is above this score. Then, the fuzzer should mark it as a required non-fuzzable parameter, and the fuzzer should obtain the value from the model. Within the configuration of the fuzzer, the following two attributes should be configurable: the score and the minimum of performed tests with this parameter.

5.2.5. Execution and evaluation

Besides executing tests and storing the responses from the SUT, the fuzzer is also responsible for updating the model (see section 5.2.4) and aborting sequences of dependent endpoints. The fuzzer aborts sequences when the previous test did not result within the HTTP 200 range. In this case, no information is available for the following requests. Without this step, the fuzzer could generate a lot of invalid requests because the state information is unavailable.

To evaluate the results of the fuzzer, a database containing the requests and responses is available. Within this database, SQL queries can be used to return rows that may contain a vulnerability. In addition, Xdebug creates log files during the execution of the fuzzer. These are available for evaluating code coverage.

5.3. Implementation

This section describes the implementations of the phases within the prototype. Also, it explains the made decisions during the different phases. This section explains the made decisions in the preparation, fuzzing and reporting phases.

Instead of implementing a REST web service fuzzer from scratch, the decision is made to extend the rest-fuzzer project from Arjan Gerritsen[15]. This fuzzer is built on Java and is based on Restler[7], an open-source tool for fuzzing. This tool already consists of a back-end and a front-end, saving developing time for setting those parts up from scratch.

This implementation is extending the rest-fuzzer with the option to perform model-based whitebox fuzzing. This model-based whitebox fuzzer will contain a stateful testing functionality to follow-up dependent endpoints from a test case. The existing fuzzer needs to be made stateful to work with the dependent endpoints. The following sections explain more about the changes made.

5.3.1. Preparation phase

When the fuzzer start, the fuzzer should set up its model. The fuzzer does this by identifying the available target locations. The model-based blackbox fuzzer contains this logic.

The result of the model-based blackbox fuzzer preparation phase is a list of available

target locations. It is essential to declare the dependencies of a target location to improve the program understanding. This is the second step within the model-based whitebox fuzzer initialisation phase. With this data available, the fuzzer creates a model. Furthermore, the start- and end times are declared based on the configured duration.

5.3.2. FUZZING PHASE

The fuzzer will go through the same steps at every fuzzing loop while the timeout does not exceed. Based on the dependencies, the fuzzer creates a series of stateful sequences.

Now the fuzzer will loop through the created sequences. First, a target location is selected, then the request is created. When this finishes, the fuzzer sends the request to the SUT. Finally, the response from the SUT is stored, and a dependent target location is selected when available.

When a sequence finishes, the following sequence starts until the timeout exceeds. The prototype repeats the steps within the fuzzing phase when the timeout not exceeds. Eventually, when the timeout exceeds, the fuzzer can finish its last call. However, no new or dependent requests are fired again at the SUT.

5.3.3. REPORTING

As mentioned in chapter 3.3, the model-based blackbox fuzzer uses Xdebug to measure the code coverage. The model-based whitebox fuzzer uses Xdebug as well. Together with the reports the rest-fuzzer creates, this provides the required metrics to compare the fuzzing methods.

5.4. VALIDATION

It is important to test the prototype on a SUT that is easy to control and monitor to validate the implementation of the prototype. As discussed in chapter 3, SutSqlI is the used SUT to test the prototype. With this SUT, the validation of the prototype is done. This section will validate the working of the prototype using the test SUT.

The validation of the prototype is done with the results of the experiments on the SUT. This validation uses the testing environment with SutSqlI. This validation uses a model-based blackbox fuzzer and a model-based whitebox fuzzer. This validation excludes the traditional whitebox fuzzer because the SUT consists of a Java [2] back-end instead of a PHP [3] back-end. This makes it impossible for the chosen traditional whitebox fuzzer to fuzz the SUT.

Figure 5.3 illustrates the HTTP status codes SutSqlI returned while executing the prototype with model-based whitebox fuzzing. With these results, it is possible to see that all requests (n=5486) which the fuzzer sent, within 60 seconds, to the SUT resulted in an HTTP 200 status code. This means that all requests are valid.

[2] https://java.com/nl/
[3] https://www.php.net/

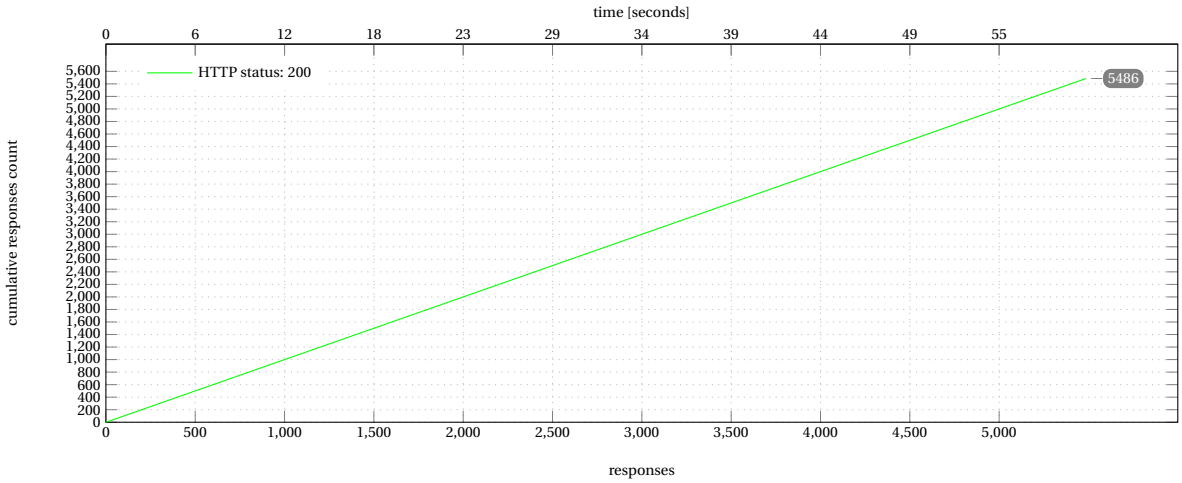

Figure 5.3: SutSqlI Test Environment: Model-based whitebox fuzzing - Responses

Figure 5.4 illustrates the HTTP status codes SutSqlI returned while executing the prototype with model-based blackbox fuzzing. With these results, it is possible to see that 51.01% of all requests (n=7998) which the fuzzer sent, within 60 seconds, to the SUT resulted in an HTTP 200 status code and are valid. The other 48.99% of the requests (n=7681) resulted in an HTTP 404 status code and are invalid.

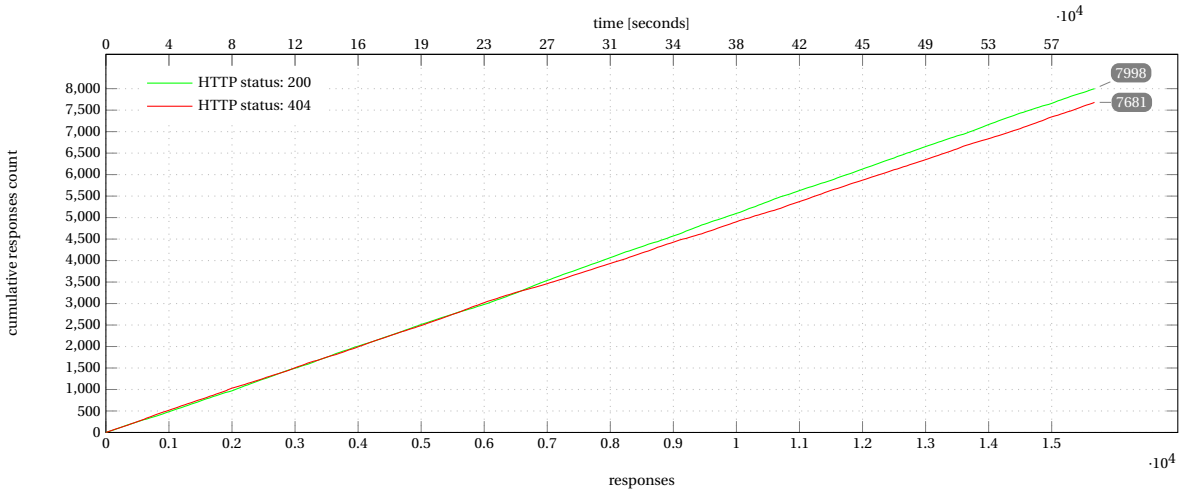

Figure 5.4: SutSqlI Test Environment: Model-based blackbox fuzzing - Responses

From these results, it is possible to see that the model-based whitebox fuzzer prototype has a higher percentage of valid requests when comparing it to the model-based blackbox fuzzer. This result is expected because the model-based whitebox fuzzer is stateful, and the model-based blackbox fuzzer is not stateful. That the model-based whitebox fuzzer is stateful helps the fuzzer with the test case generation. Based on the results of previous dependent requests, the fuzzer generates a new test case. This test case uses obtained parameters to create the input for the request. Based on figure 3.1, an example can be made. This example uses two endpoints, `GET /rest/comments/{id}` and its dependency `POST /rest/comments` and the main focus here is to test `GET /rest/comments/{id}`. The fuzzer uses the returned id from `POST /rest/comments` to get an existing object from `GET /rest/comments/{id}` when using a stateful approach. Random values will be placed for id when using a stateless approach. This results in many HTTP 404 status codes because the provided id may not be attached to an existing object.

Both fuzzers did not find a vulnerability during the execution of the test. However, since the model-based whitebox fuzzer is stateful, more valid inputs can be generated. This could help to find vulnerabilities within the SUTs faster.

5.5. Results

This section elaborates on the results of the experiments. These results provide an answer for RQ3. For the experiments, the SUTs WooCommerce and Drupal are selected. The other described potential SUTs made it challenging to find an OpenAPI specification that was also understandable for machines. Most of them only had a public specification that was readable for humans. For example, Magento uses ReDoc[4] to convert the specification into a more user-friendly document. Only WooCommerce and Drupal were possible within the given time limits. For WooCommerce, it was possible to measure the code coverage. However, the fuzzer does not measure the code coverage of Drupal. Drupal uses imported modules which made it hard to modify them to implement the Xdebug debugger within them.

5.5.1. Model-based blackbox fuzzing

This section covers the testing results of the model-based blackbox fuzzer. With the model-based blackbox fuzzer, several tests have been performed. Each SUT is tested three times for 2 hours. Underneath, the paragraphs display the testing results for these tests. The results will cover the number of executed tests, the status code categorisation, code coverage and the discovered vulnerabilities.

Requests The first metrics, the number of executed tests and the status code categorisation are obtainable from analysing the test results of the experiments. Below, figures 5.5, 5.6, 5.7, 5.8, 5.9 and 5.10 illustrate the model-based blackbox fuzzing runs on the two SUTs.

When looking at the experiments on the first SUT, it is possible to see that the number of valid requests is relatively low in the first three figures. The three executed experiments produce in total 7446 requests. From these 7446 requests, only 176 (2.36%) of the requests were valid; responses with HTTP status codes in the 200 range. This percentage is the ratio of responses with status code 200 (n=176) to the total number of requests (n=7446).

The majority of the requests are invalid. In total, 7067 (94.91%) of the requests did produce a response with an HTTP status code in the 400 range. This percentage is the ratio of responses with status code 404 (n=3644), 400 (n=2475), 401 (n=948) to the total number of requests (n=7446). The remaining 203 (2.73%) requests did result in a response with an HTTP status code in the 500 range. This percentage is the ratio of responses with status code 501 (n=203) to the total number of requests (n=7446).

When looking at the experiments on the second SUT, it is possible to see that the number of valid requests is relatively low in the last three figures compared to the other status codes. These three executed experiments produce in total 30518 requests. From these 30518 requests, 5635 (18.46%) of the requests were valid; responses with HTTP status codes in the 200 range. This percentage is the ratio of responses with status code 200 (n=5635) to the total number of requests (n=30518).

[4] https://github.com/Redocly/redoc

The majority of the requests are invalid. However, 24883 (81.54%) of the requests did produce a response with an HTTP status code in the 400 range. This percentage is the ratio of responses with status code 404 (n=17355) and 415 (n=7528) to the total number of requests (n=30518).

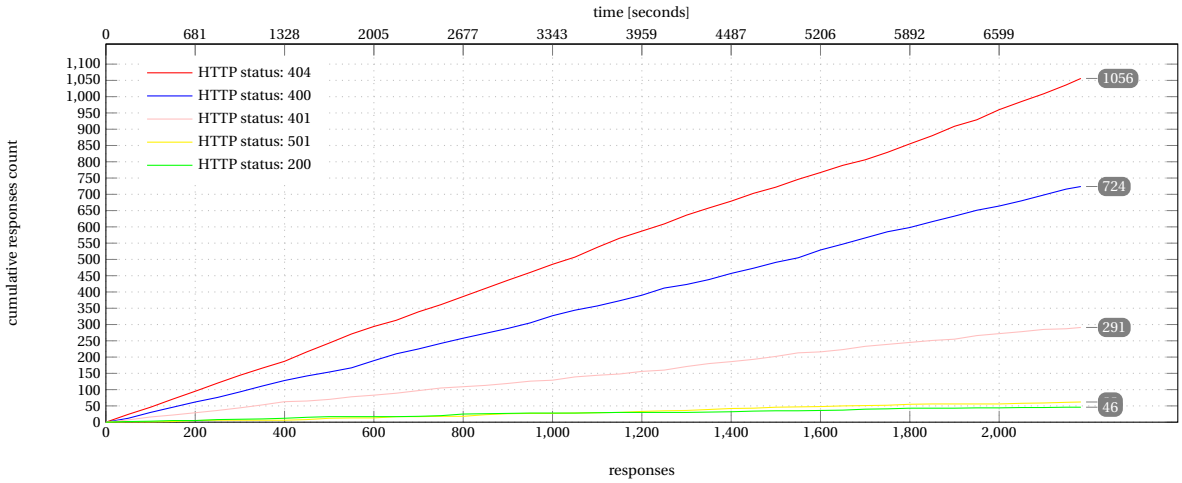

Figure 5.5: WooCommerce: Model-based blackbox fuzzing - Responses Run 1

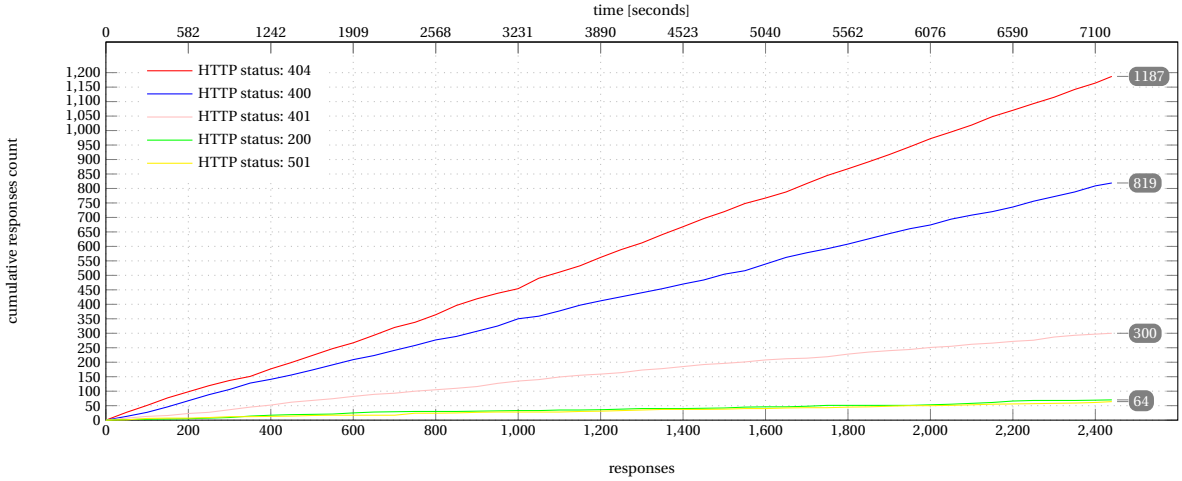

Figure 5.6: WooCommerce: Model-based blackbox fuzzing - Responses Run 2

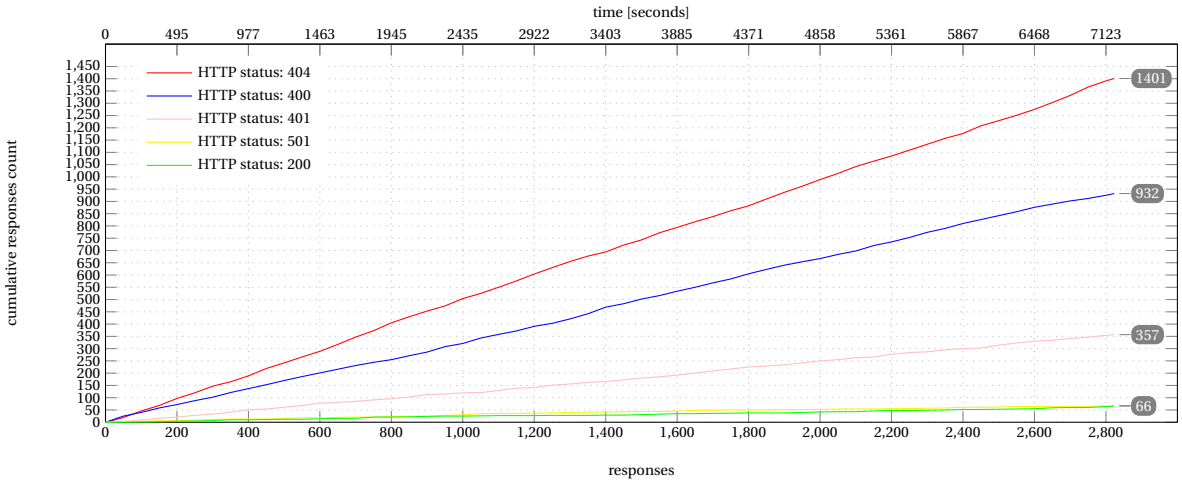

Figure 5.7: WooCommerce: Model-based blackbox fuzzing - Responses Run 3

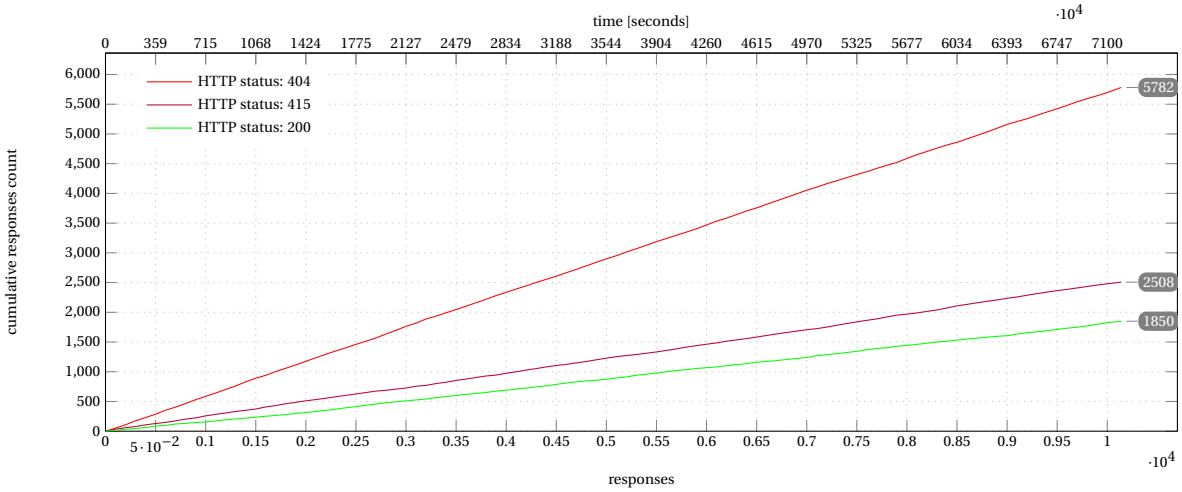

Figure 5.8: Drupal: Model-based blackbox fuzzing - Responses Run 1

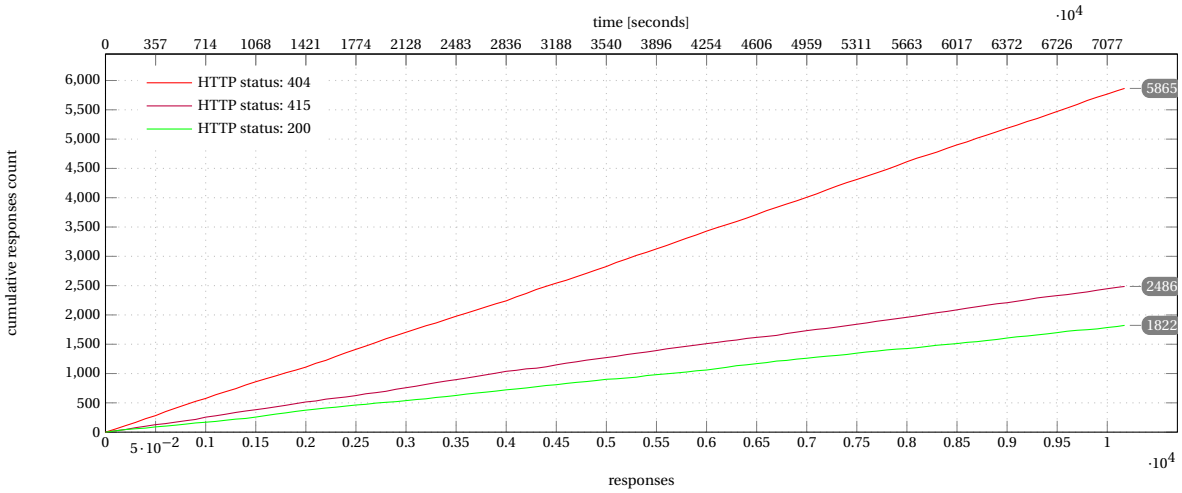

Figure 5.9: Drupal: Model-based blackbox fuzzing - Responses Run 2

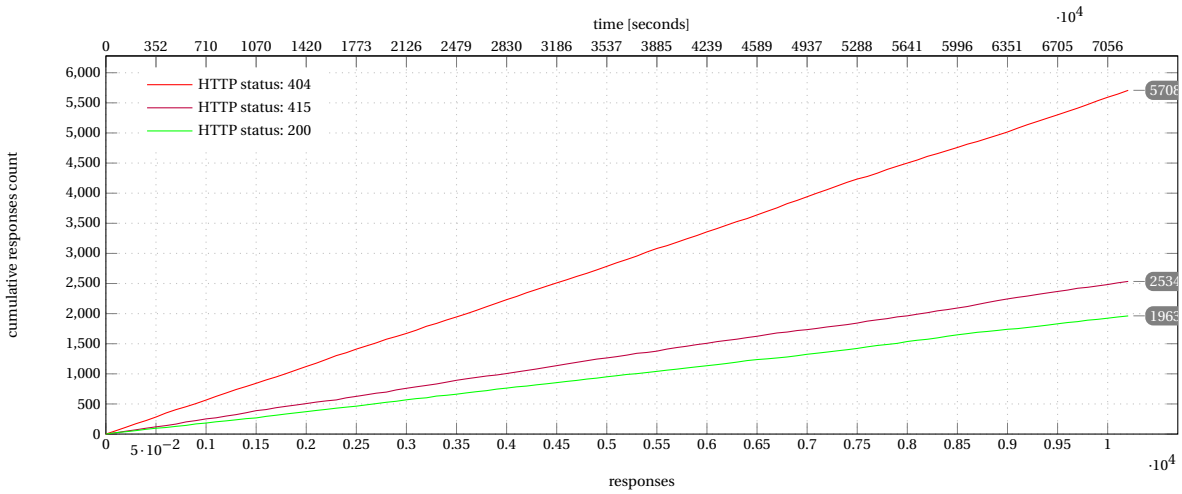

Figure 5.10: Drupal: Model-based blackbox fuzzing - Responses Run 3

Code coverage The model-based blackbox fuzzer could not calculate the code coverage for the REST web service in the Drupal project. As a result, only data from the REST web service in the WooCommerce project could be gathered. Unfortunately, Drupal uses modules which makes it hard to modify them when setting up the SUT. For this reason, the fuzzer does not track the code coverage for Drupal. Figures 5.11, 5.12, and 5.13 illustrate the code coverage for the WooCommerce project.

From the results, it is possible to see that the total executed lines of code initially go up sharply and then stagnate at a certain point at the start. In Run 1, this happens after 1400 requests. In run 2 and run 3, this already happens after 800 requests. In all three cases, the total number of executed lines is 789. This is 11.48% of the total lines of code (n = 6889). Although the fuzzer executes 11.48% of all the source code, the SUT executes no endpoint (n=0) entirely. This could happen when the input does not execute all paths within the implementation of the endpoint.

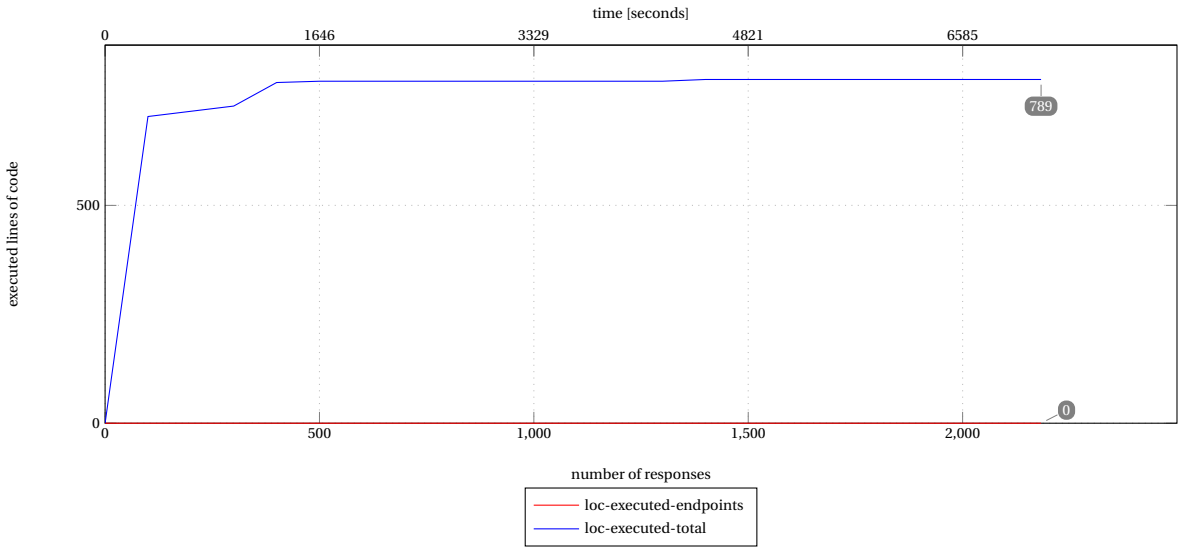

Figure 5.11: WooCommerce: Model-based blackbox fuzzing - Coverage Run 1

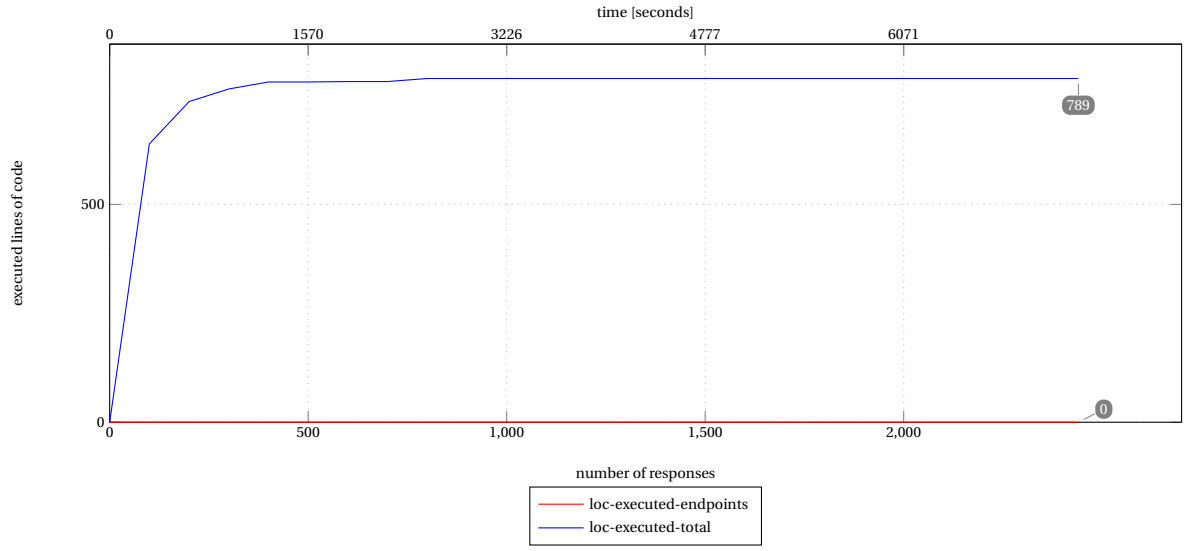

Figure 5.12: WooCommerce: Model-based blackbox fuzzing - Coverage Run 2

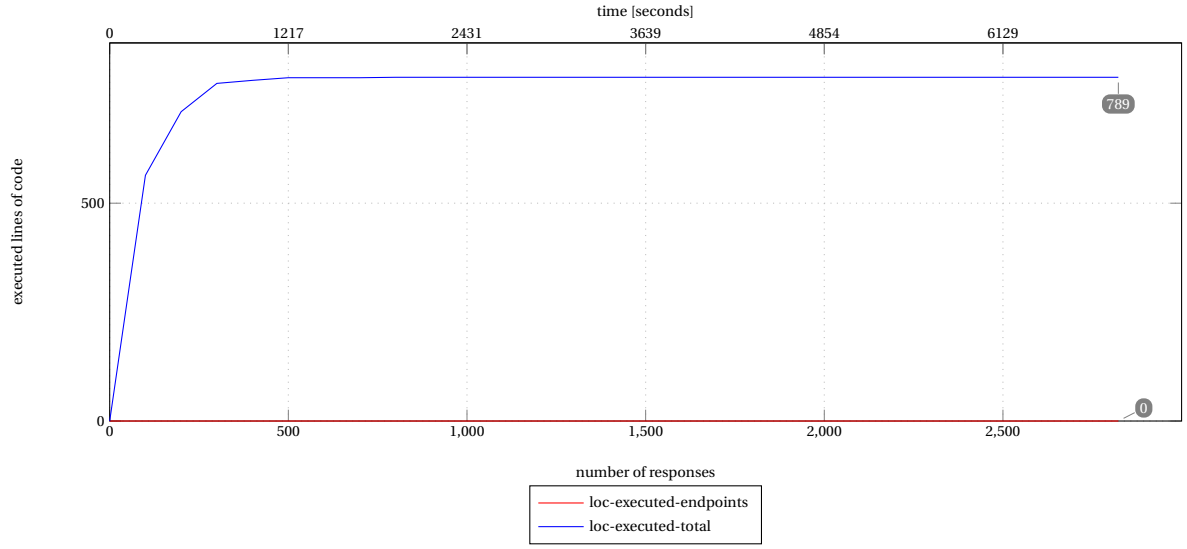

Figure 5.13: WooCommerce: Model-based blackbox fuzzing - Coverage Run 3

Vulnerabilities With model-based blackbox fuzzing, only one vulnerability is discovered within the selected vulnerability types. The discovered vulnerability is the HTTPS vulnerability. This vulnerability disappears when installing an SSL certificate [5] on the server. In this case, the SUT can use HTTPS, and the specification includes HTTPS in the schemes array within the specification.

Furthermore, the fuzzer did not trigger another vulnerability from the other selected vulnerability types:

- **Access Control** - The project configurations do not include credentials. As a result, the WooCommerce SUT returns a lot of 401 status codes. For the Drupal SUT, the fuzzer used the default credentials.

[5] https://www.ssl.com/

- **Injection** - By scanning the responses for common injection inputs [6], none was discovered, leading to a vulnerability within both SUTs.
- **Validate content types** - For both SUTs, this vulnerability did not occur. This was checked by looking at the requests that had a 406 or 415 status code.
- **Error handling** - The SUTs do not include error handling vulnerabilities. None of the responses with a 500 status code did include traces or good, useful feedback.
- **Sensitive information in HTTP Requests** - No sensitive information is added within the path URL for both SUTs. When scanning the path variables, the query contained words as 'password', 'creditcard', 'key' and 'license'.

5.5.2. TRADITIONAL WHITEBOX FUZZING

This section covers the testing results of the traditional whitebox fuzzer. With the traditional whitebox fuzzer, several tests have been performed. Each SUT is tested three times until completion. Below, the testing results are displayed for these tests. The results will cover the number of executed tests, the status code categorization, code coverage and the discovered vulnerabilities.

Requests The traditional whitebox fuzzer runs until completion. When the fuzzer finishes, all responses can be analysed in an individual file. The fuzzer executed 2074 tests for the REST web service in the WooCommerce project. When merging the response files (.txt - text documents) with the command prompt, with the command "copy *_Response.txt combined.txt", the combined.txt file can be analyzed for the HTTP status codes. In total, 2074 tests are valid from the 2074 executed tests. All responses include the HTTP 200 status code. Figure 5.14 visualises these results.

The traditional whitebox fuzzer performed 915 tests on the REST web service in the Drupal project. The combined response file, generated using the same command, only contains HTTP 403 status codes. None of the executed tests was a valid test for the REST web service in the Drupal project. Figure 5.15 visualises this. The reason for this could be that Drupal requires authorization. This is something the traditional whitebox fuzzer does not include.

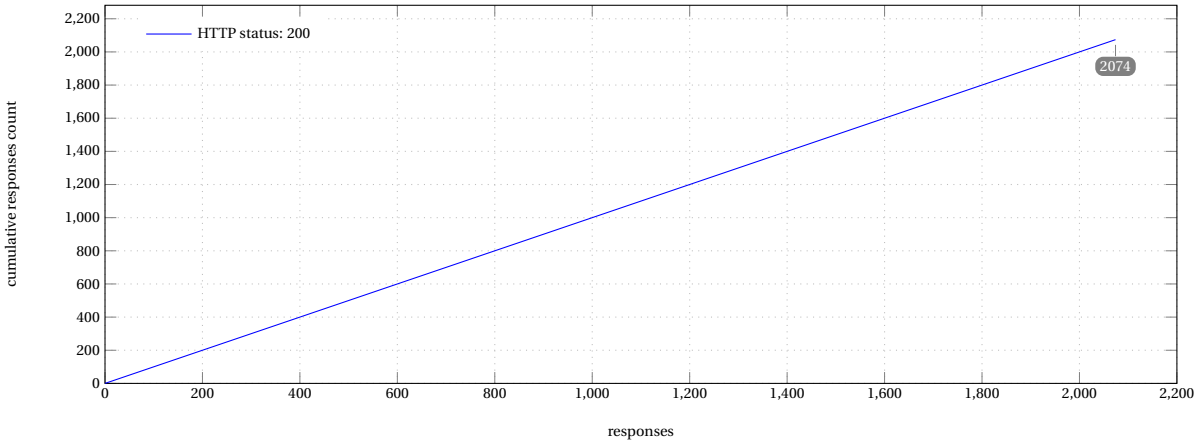

Figure 5.14: WooCommerce: Tradition whitebox fuzzing - Responses

[6] https://github.com/payloadbox

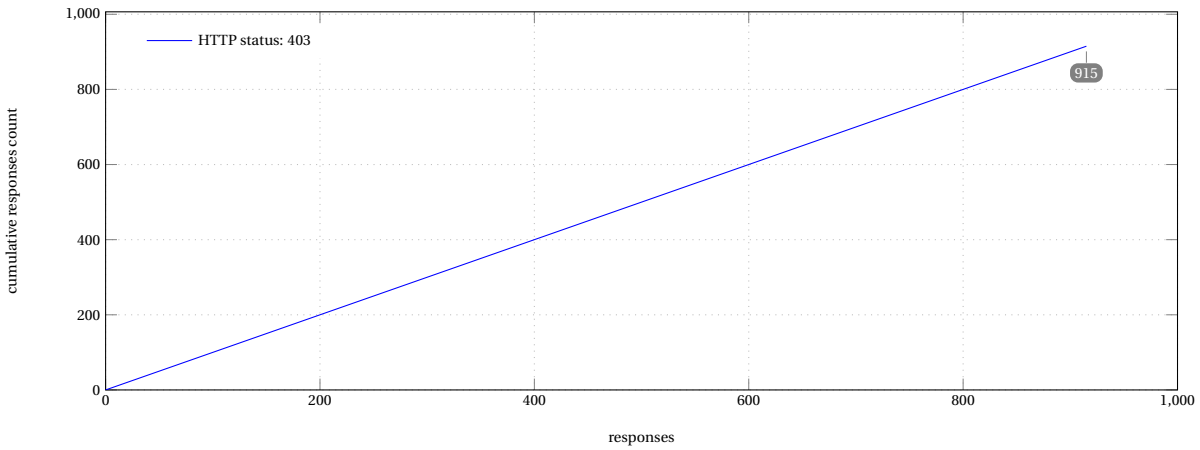

Figure 5.15: Drupal: Tradition whitebox fuzzing - Responses

Code coverage The traditional whitebox fuzzer could calculate the code coverage of the REST web service in the WooCommerce project and the Drupal project.

From the results, it is possible to see that every scan plugin was executed on the same lines of code. Every scan plugin reaches a code coverage of only 1.69% within the REST web services from the WooCommerce project. Figure 5.16 illustrates this. This results in a total of 118 executed lines of code, as shown in figure 5.17. For the REST web services in the Drupal project, the fuzzer achieves a code coverage of 0%. This results in a total of 0 executed lines of code, as shown in figure 5.18. This is expected based on the responses. All requests did return an HTTP 403 status code. This means that using the endpoint is forbidden. Based on that result, the obtained code coverage of 0% is explainable because the fuzzer did not trigger an endpoint to execute.

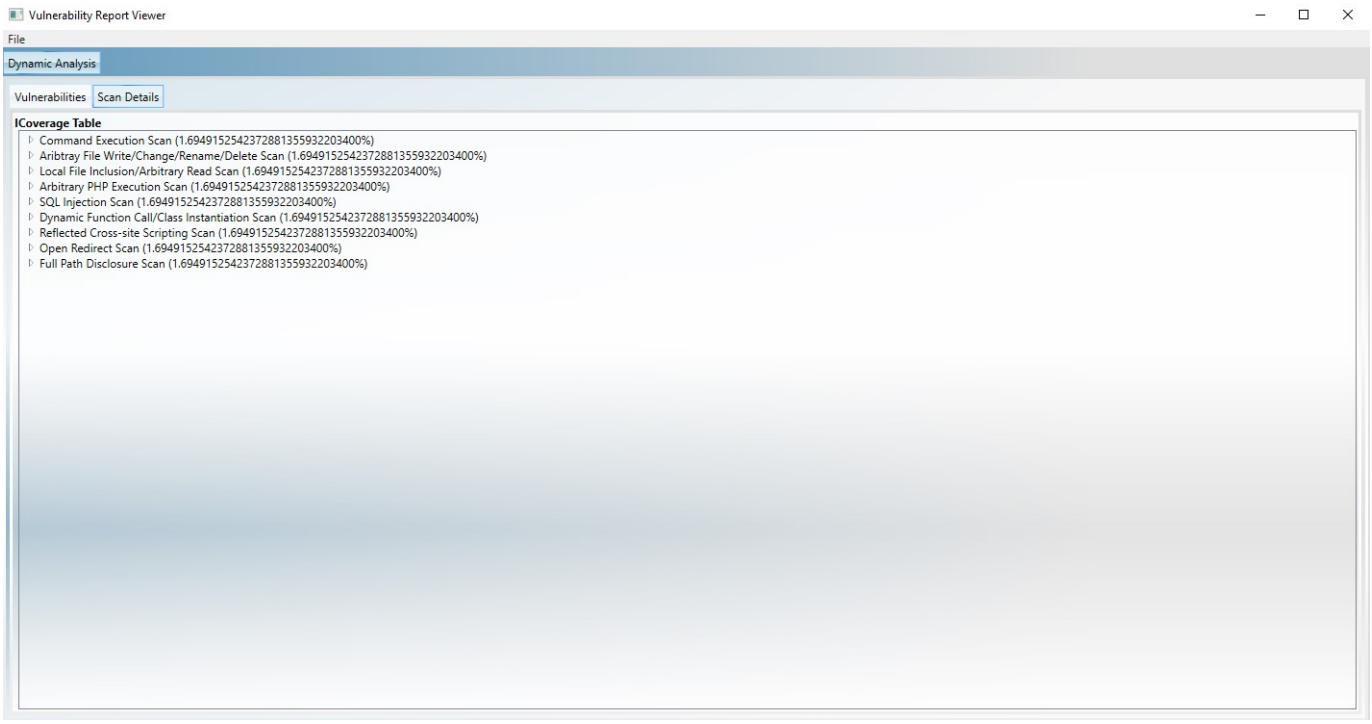

Figure 5.16: WooCommerce: Traditional whitebox fuzzing - Code coverage results

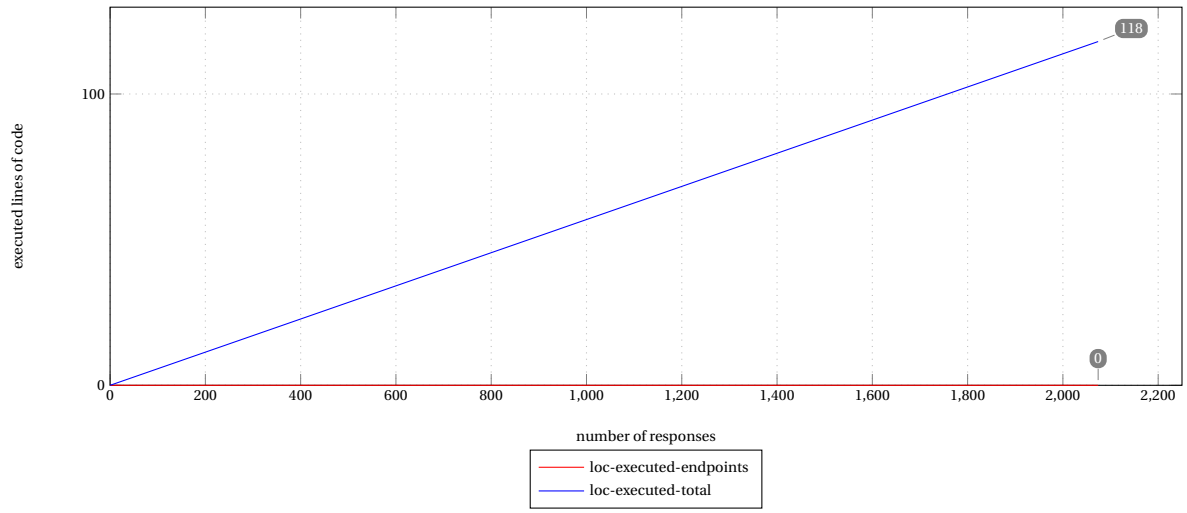

Figure 5.17: WooCommmerce: Tradition whitebox fuzzing - Coverage

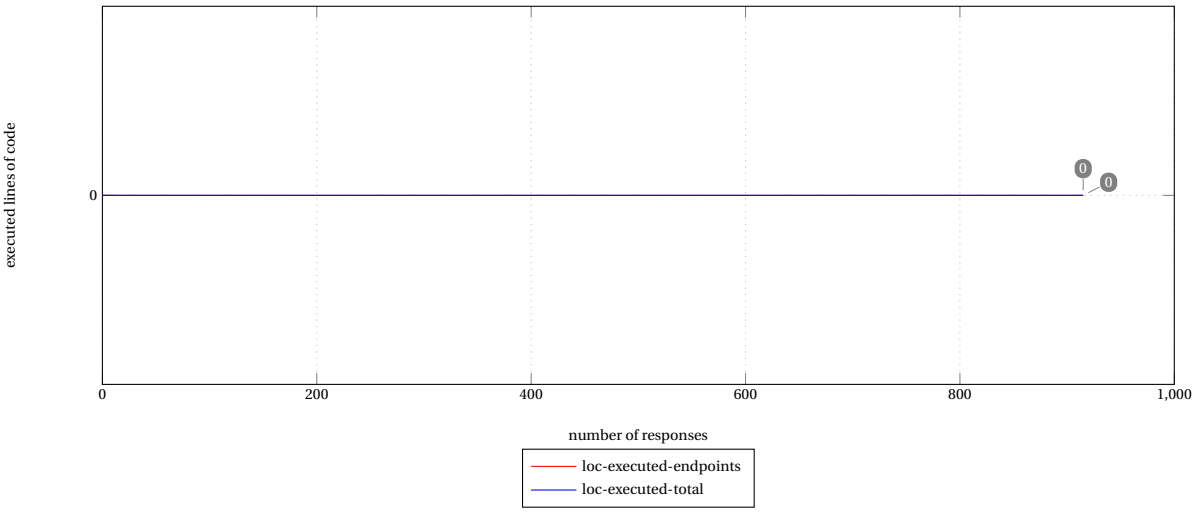

Figure 5.18: Drupal: Tradition whitebox fuzzing - Coverage

Vulnerabilities During the experiments, the traditional whitebox fuzzer used all available scan plugins. With these plugins, the traditional whitebox fuzzer discovered the same 58 vulnerabilities in the REST web service from WooCommerce. All these vulnerabilities are of the Full Path Disclosure type. Within the responses, an error occurs since a file is not found. Within the body of the response, the SUT exposes a full system path. Figure 5.19 contains an example of one response. The figure is responding with an exception, and the error message contains two times the system path `C:\xampp\htdocs\wordpressWC\wp-includes\rest-api\search\class-wp-rest-response.php`. The CVE database does not list this vulnerability, but this vulnerability has existing remediation. If in the php.ini file the property "display_errors" is set to "off", this vulnerability should not occur anymore [7].

[7] https://www.acunetix.com/vulnerabilities/web/wordpress-full-path-disclosure/

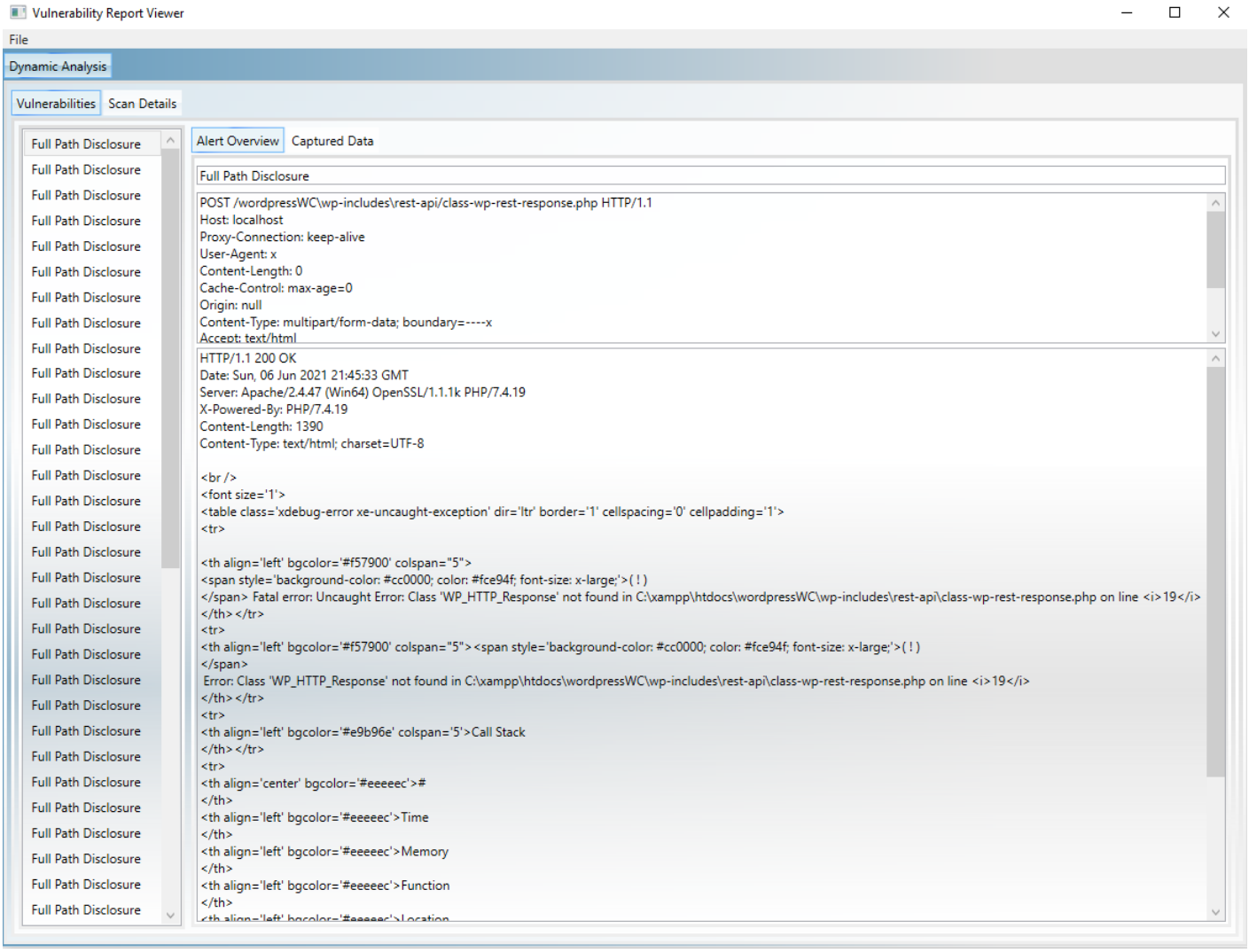

Figure 5.19: WooCommerce: Traditional whitebox fuzzing - Vulnerability results

Github[8] hosts the documents containing the combined responses from the SUTs. Within the responses, the vulnerabilities can be found.

5.5.3. Model-based whitebox fuzzing

This section covers the testing results of the model-based whitebox fuzzer. With the model-based whitebox fuzzer, several tests have been performed. Each SUT is tested three times for 2 hours. Underneath, the paragraphs display the testing results for these tests. The results will cover the number of executed tests, the status code categorisation, code coverage and the discovered vulnerabilities.

Requests The number of executed tests and the status code categorisation is derived from analysing the test results of the executed experiments. Below, figures 5.20, 5.21, 5.22, 5.23, 5.24 and 5.25 illustrate the model-based whitebox fuzzing runs on the two SUTs.

When looking at the experiments on the first SUT, it is possible to see that the number of valid requests is relatively high in the first three figures. The three executed experiments

[8] https://github.com/JasonKleuskens1/PhpVulnerabilityHunter-experiments

produce in total 14343 requests. From these 14343 requests, 8010 (55.85%) of the requests were valid; responses with HTTP status codes in the 200 range. This percentage is the ratio of responses with status code 200 (n=6509) and status code 201 (n=1501) to the total number of requests (n=14343).

The majority of the requests are valid. However, the remainder is invalid. In total, 6333 (44.15%) of the requests did produce a response with an HTTP status code in the 400 and 500 range. This percentage is the ratio of responses with status code 404 (n=2013) and status code 400 (n=3164) to the total number of requests (n=14343). Thus, the 400 range HTTP status codes cover 36.09% of the 44.14%. The remaining 1159 (8.06%) requests did result in a response with an HTTP status code in the 500 range. This percentage is the ratio of responses with status code 501 (n=1156) and status code 500 (n=3) to the total number of requests (n=7446).

When looking at the experiments on the second SUT, it is possible to see that the number of valid requests is lower in the last three figures compared to the other status codes than it was for the WooCommerce project. These three executed experiments produce in total 30322 requests. From these 30322 requests, 9864 (32.53%) of the requests were valid; responses with HTTP status codes in the 200 range. This percentage is the ratio of responses with status code 200 (n=9864) to the total number of requests (n=30322).

The majority of the requests are invalid. In total, 20458 (67.47%) of the requests did produce a response with an HTTP status code in the 400 range. This percentage is the ratio of responses with status code 405 (n=12433), status code 404 (n=7596), and 415 (n=429) to the total number of requests (n=30322).

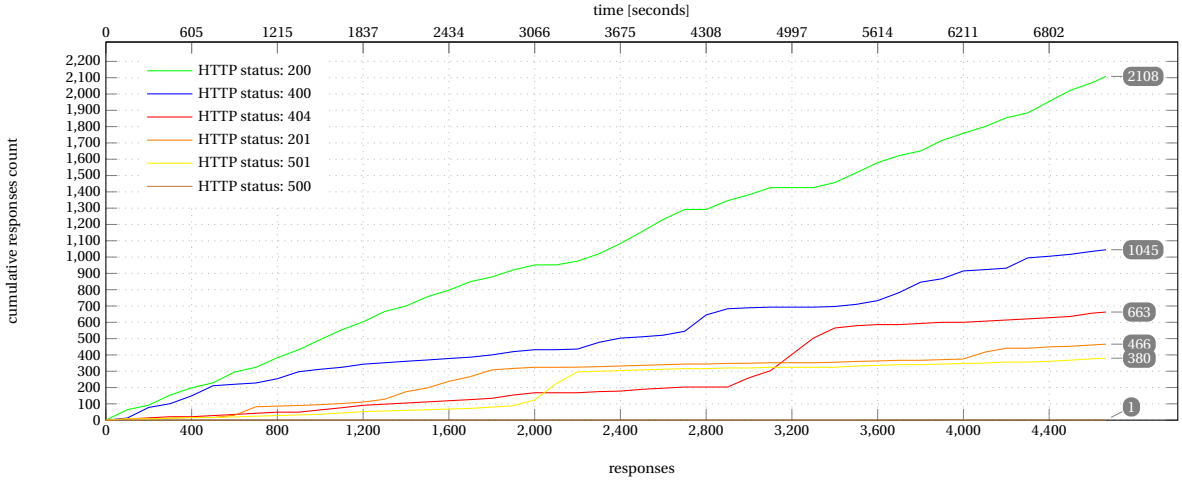

Figure 5.20: WooCommerce: Model-based whitebox fuzzing - Responses Run 1

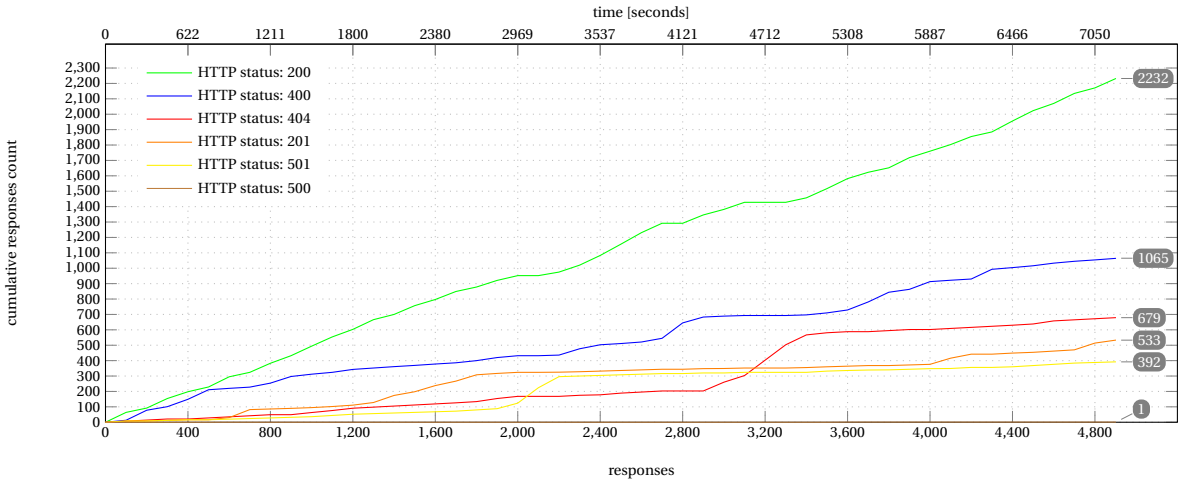

Figure 5.21: WooCommerce: Model-based whitebox fuzzing - Responses Run 2

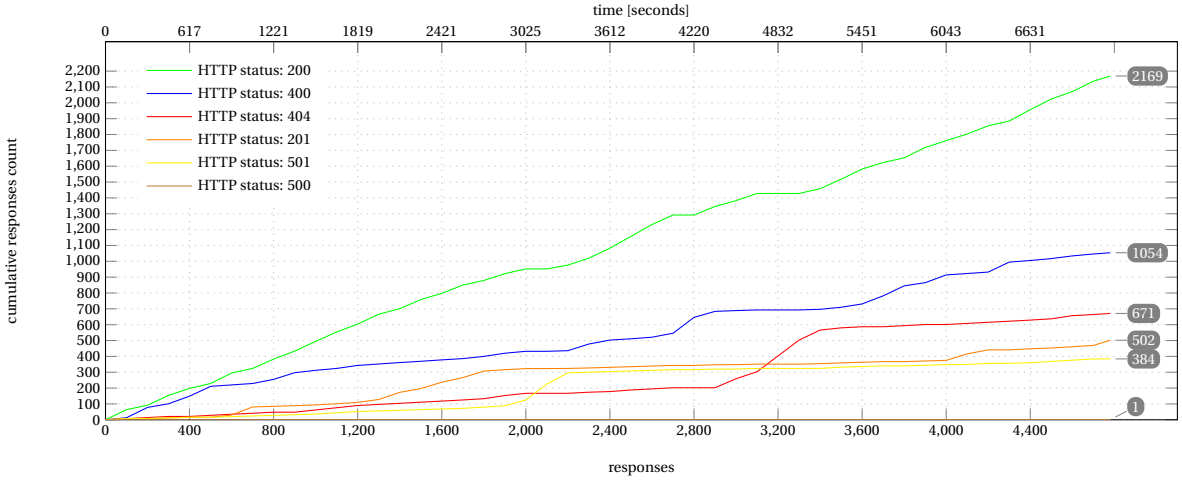

Figure 5.22: WooCommerce: Model-based whitebox fuzzing - Responses Run 3

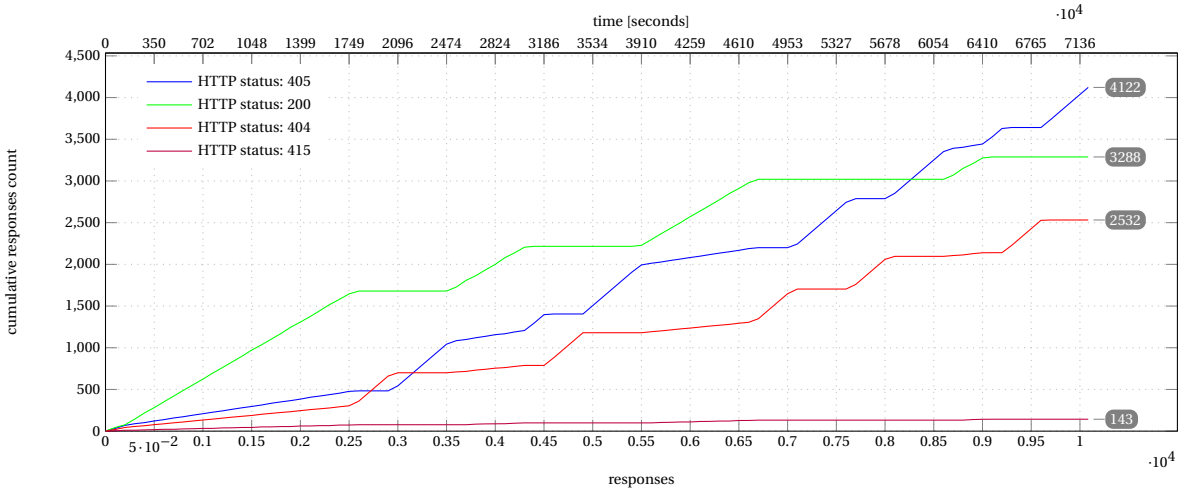

Figure 5.23: Drupal: Model-based whitebox fuzzing - Responses Run 1

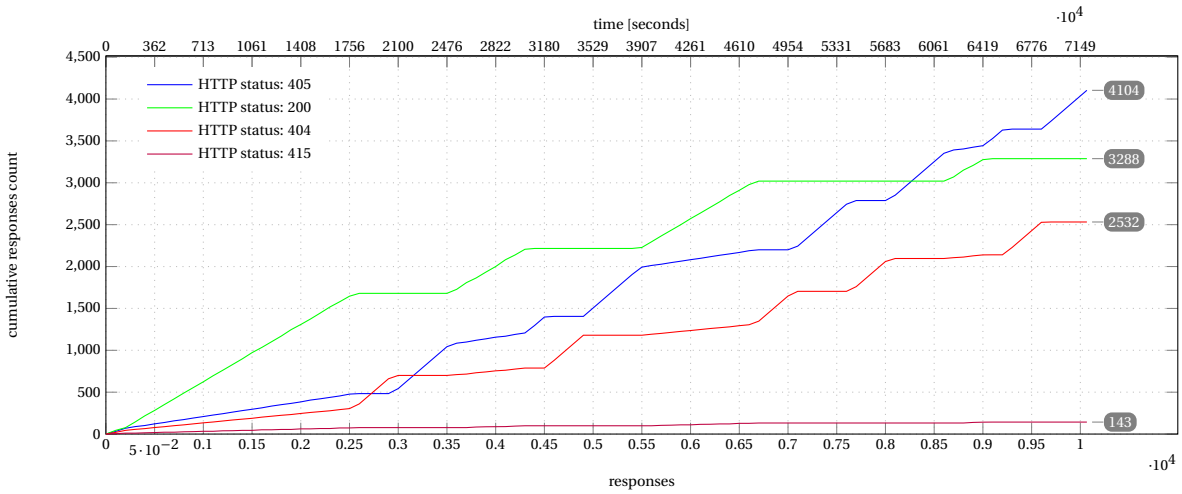

Figure 5.24: Drupal: Model-based whitebox fuzzing - Responses Run 2

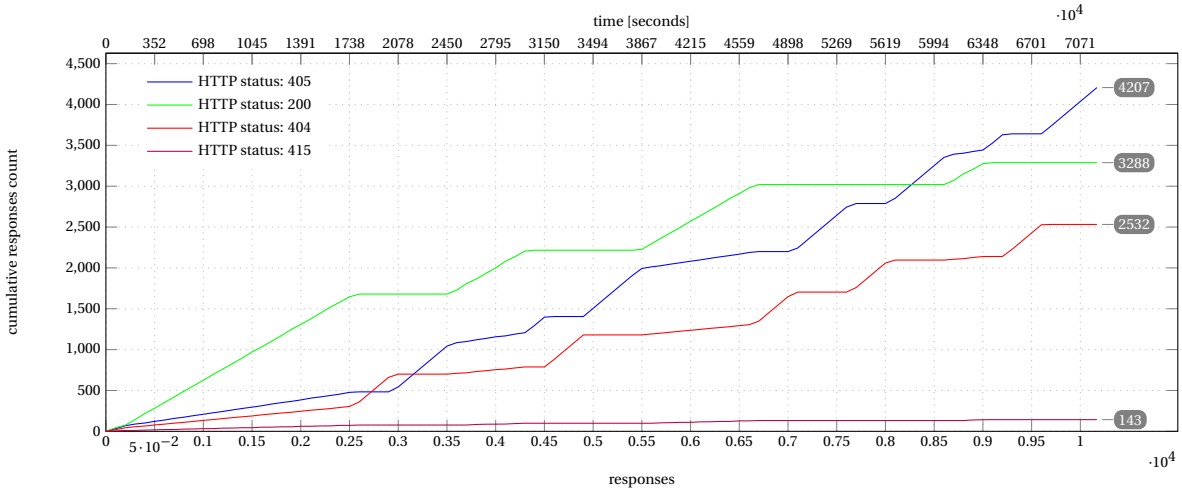

Figure 5.25: Drupal: Model-based whitebox fuzzing - Responses Run 3

Code coverage The model-based whitebox fuzzer could not calculate the code coverage for the REST web service in the Drupal project; this was the same as for the model-based blackbox fuzzer. As a result, only data from the REST web service in the WooCommerce project could be gathered. This is explainable because the model-based whitebox fuzzer extends the model-based blackbox fuzzer and uses the same techniques. Figures 5.26, 5.27, and 5.28 illustrate the achieved code coverage for the WooCommerce SUT.

From the results, it is possible to see that the total executed lines of code initially go up sharply and then stagnate at a certain point at the start. From the halfway point, the fuzzer does not execute a new line of code in the SUT. In run 1, run 2 and run 3, this happens after 1800 requests. In all three cases, the total number of executed lines is 2339. This is 33.59% of the total lines of code (n=6889). Although this number is relatively high, still no endpoint (n=0) is executed entirely. This could happen when not all paths within an endpoint are executed.

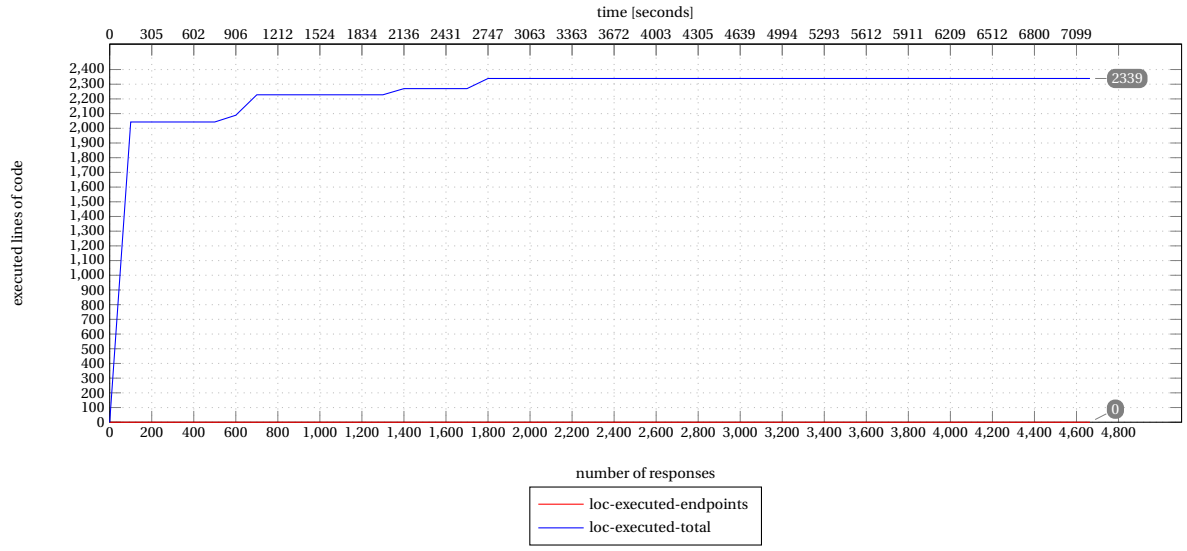

Figure 5.26: WooCommerce: Model-based whitebox fuzzing - Coverage Run 1

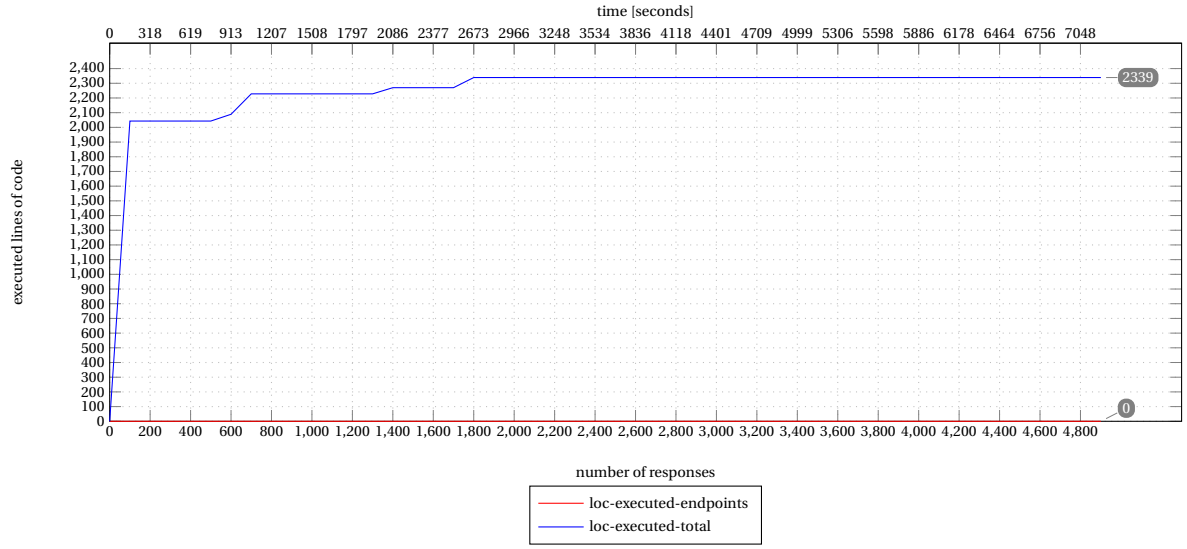

Figure 5.27: WooCommerce: Model-based whitebox fuzzing - Coverage Run 2

Vulnerabilities With model-based whitebox fuzzing, only one vulnerability is discovered within the selected vulnerability types. The discovered vulnerability is the HTTPS vulnerability. The experiments are not executed on a server that was provided with an SSL certificate. However, this vulnerability has existing remediation. This vulnerability disappears when installing an SSL certificate on the server. In this case, the SUT can use HTTPS, and the specification includes HTTPS in the schemes array within the specification.

Furthermore, the fuzzer did not trigger another vulnerability from the other selected vulnerability types:

- **Access Control** - The project configurations provide the credentials. The SUT authorises the fuzzer based on these credentials.

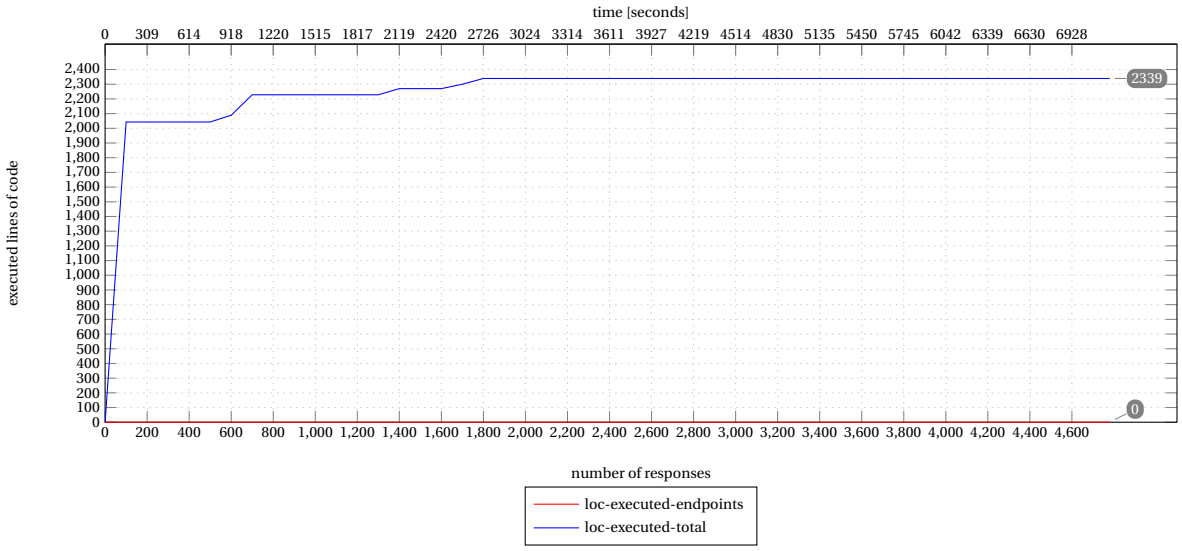

Figure 5.28: WooCommerce: Model-based whitebox fuzzing - Coverage Run 3

- **Injection** - By scanning the responses for common injection inputs [9], none was discovered, leading to a vulnerability within both SUTs.
- **Validate content types** - For both SUTs, this vulnerability did not occur. This was checked by looking at the requests that had a 406 or 415 status code.
- **Error handling** - The SUTs do not include error handling vulnerabilities. None of the responses with a 500 status code did include traces or useful feedback.
- **Sensitive information in HTTP Requests** - No sensitive information is added within the path URL for both SUTs. When scanning the path variables, words as 'password', 'creditcard', 'key' and 'license' were used.

5.5.4. SUMMARY

This section shows the results of the experiments in a table. This table helps to compare the fuzzing methods based on the results from the used SUTs.

Table 5.1: Summary experimental results

Fuzzer	SUT	Vulnerabilities	Executed tests	Valid requests	Code coverage
Model-based blackbox fuzzer	WooCommerce	1	7446	2.36%	11.48%
Model-based blackbox fuzzer	Drupal	1	30518	18.46%	-
Traditional whitebox fuzzer	WooCommerce	58	2074	100%	1.69%
Traditional whitebox fuzzer	Drupal	0	915	0%	0.0%
Model-based whitebox fuzzer	WooCommerce	1	14343	44.14%	33.59%
Model-based whitebox fuzzer	Drupal	1	30322	32.53%	-

[9] https://github.com/payloadbox

6

DISCUSSION, CONCLUSION AND FUTURE WORK

This research focuses on how a model-based whitebox fuzzing approach can be applied to REST web services to detect vulnerabilities. This research provides an answer to this question by answering the three subquestions earlier mentioned in chapter 3.

- **RQ1:** What types of vulnerabilities can be detected in REST web services with model-based whitebox fuzz testing?
- **RQ2:** How can the model-based whitebox fuzzing approach on program binaries be applied to REST web services?
- **RQ3:** How does model-based whitebox fuzzing perform on REST web services compared to traditional whitebox fuzzing and model-based blackbox fuzzing?

The remainder of this chapter provides the answers to these research questions. Furthermore, it makes conclusions and provides directions for future work.

6.1. VULNERABILITIES IN REST WEB SERVICES

This research uses literature to get more insight into the known vulnerabilities for REST web services. Section 4.1 provides a list of all vulnerabilities for REST web services. This research also shows that the model-based whitebox fuzzer could detect a number of these potential vulnerabilities. However, the fuzzer can not detect all vulnerabilities from this list due to the made design decisions. From the vulnerabilities, the following vulnerabilities are fuzzable based on the chosen techniques:

- HTTPS
- Access control
- Injection attacks
- Validate content types
- Error handling
- Sensitive information in HTTP Requests.

6.2. Model-based whitebox fuzzing approach

Chapter 5 provides information about how the model-based fuzzing approach can be used on REST web services. The proposed approach removes some steps from the original model-based whitebox fuzzing approach because these steps can be simplified with the obtained model. This makes the approach less complex than the original. In addition, the fact that the prototype uses the OpenAPI specification rather than the program binaries helps reduce the complexity for the most part.

During the scheduling phase, the fuzzer identifies the critical locations by looking at the available endpoints. The original approach uses a model-based search to find available endpoints close to the previously tested endpoint. However, it is enough to look at the dependent endpoints to find the critical locations for fuzzing REST web services. These dependent endpoints are the closest to the previously tested endpoint and are obtainable from the OpenAPI specification of the SUT.

During the input generation phase, the fuzzer uses file cracking, file stitching, and file repair in the initial model-based whitebox fuzzing approach. First, the fuzzer uses file cracking to extract potential fragments from previous responses within the model that the fuzzer could use to generate valid inputs. Then, based on the state of each parameter, file stitching uses fragments to place values from the model on the matching locations within the request. Finally, the fuzzer uses file repair to ensure that the generated input is a valid object. The same approach would work for the prototype. First, the needed parameters are selected. This followed by creating a new request to fire at the SUT.

The current approach removes some steps from the original model-based whitebox fuzzing approach. Therefore, it is not possible to gain knowledge on the crucial ifs. Source code is required to gain this knowledge, but the prototype will only use the OpenAPI specification during run time. Furthermore, each run can generate a single input because the OpenAPI specification provides this for every endpoint.

The developed model-based whitebox fuzzing prototype can generate valid inputs for the testing SUT SutSqlI. In total, 100% of the requests did result in an HTTP status code in the 200 range. This is explainable because the fuzzer detects the sequences based on the found dependencies between the endpoints. The fuzzer creates valid inputs for the dependent endpoints with the obtained values within the responses. However, the developed prototype did not discover the SQL injection vulnerability. This is explainable because no input was generated that should trigger a SQL injection. The fuzzer can create inputs that trigger SQL injection vulnerabilities, but for this to happen, the fuzzer has to create these inputs by chance. Only for parameters with a string type, this can happen. These parameters are most often fuzzable, and that means that the fuzzer generates a random string.

6.3. Fuzzers comparison

This research obtains results from the executed experiments with the model-based whitebox, model-based blackbox, and traditional whitebox approaches. The results come from experiments with the SUTs WooCommerce and Drupal.

Firstly, the fuzzers did not find a new vulnerability when performing the experiments; the fuzzer only discovered known vulnerabilities with existing remediation. The reason for this could be the large, well-developed SUTs used during this research. New projects could contain vulnerabilities that are easier to fuzz than for these well-known SUTs.

Secondly, the results with the model-based blackbox fuzzer show that the model-based blackbox fuzzer was able to send valid requests to the SUTs. However, the number of valid requests in the 200 range is low for both SUTs, respectively 2.36% and 18.46%. Most of the time, the SUTs respond with an HTTP status code within the 400 range, often the 404 HTTP status code. This means that the server could not find the requested resources. In addition, the fuzzer achieves a low code coverage percentage of 11.48% for WooCommerce, and the fuzzer only detected one vulnerability. These results are explainable because the fuzzer only generates values that fit the requested format. However, the fuzzer does not look if the requested resource is available. This means that functions within the source are potentially not fully executed when the requested resource is not available. This also explains the results in figures 5.5 until 5.10. Every time a GET, PUT or DELETE request is sent with a random identifier, the chances are that the SUT returns a 400 related HTTP status code.

Thirdly, the results from the traditional whitebox fuzzer show that the traditional whitebox fuzzer only was able to perform valid requests on WooCommerce. In total, 100% of the requests the fuzzer made were valid requests in the 200 range. However, the fuzzer could not perform a valid request on Drupal. All requests had HTTP status codes in the 400 range. Besides those results, the achieved code coverage is low for both SUTs, respectively 1.69% and 0.0%. These results are explainable because the fuzzer does not contain any specific information for the connection with the endpoints, e.g., API keys. The fuzzer obtains all information that is available from the source code. In addition, the traditional whitebox fuzzer was able to detect 58 vulnerabilities where a full system path was exposed.

Fourthly, the results from the model-based whitebox fuzzer show that the model-based whitebox fuzzer was able to send valid requests to both SUTs. The number of valid requests is low, but compared with the other fuzzers, 44.14% and 32.53% is relatively high. In addition, the results of the responses are not linear as with the model-based blackbox fuzzer. Furthermore, the achieved code coverage is relatively high, with 33.59% for WooCommerce compared to the other tested fuzzers. However, the number of discovered vulnerabilities is 1. In this case, only the HTTPS vulnerability is discovered. The results from the model-based whitebox are explainable. The proposed model-based whitebox fuzzer approach extends the model-based blackbox fuzzer with the option to keep to be stateful to access deeper parts of the SUT. This results in that dependent endpoints use the obtained response from older requests to create a valid request that access deeper parts of the SUT. This explains why the results are not linear and also makes it possible to find the vulnerabilities within the deeper parts of the SUT. However, with the executed experiments, the number of discovered vulnerabilities did not increase.

Fifthly, in the study of Gerritsen [15], the model-based dictionary fuzzer is compared with the research of Takanen et al. [36] to define if the fuzzer can be named capable of detecting vulnerabilities. Based on the data from this research, the fuzzers achieve a code coverage between 20% and 42%, and their efficiency is between 10% and 80% [36]. The developed model-based whitebox fuzzer achieves a code coverage of 33.59% and an efficiency between 32.53% and 44.14%. Based on these results, it seems that the developed model-based whitebox fuzzer is capable of detecting vulnerabilities.

Finally, the test results show that the model-based whitebox fuzzer is the best option for fuzzing REST web services; compared with model-based blackbox fuzzing and traditional whitebox fuzzing. This is because the model-based whitebox fuzzer can send more valid

requests and gain higher code coverage than model-based blackbox fuzzing and traditional whitebox fuzzing. In addition, the model-based whitebox fuzzer performed the strongest since it was stateful. With the stored information in the stateful fuzzer, the fuzzer could create more valid requests that access deeper locations within the endpoints that can only be accessed with valid existing inputs.

6.4. Conclusion

In conclusion, the products of this research are a proposal for a model-based whitebox fuzzing approach for fuzzing REST web services, a prototype capable of model-based whitebox fuzzing on REST web services based on the proposed approach and an evaluation of the effectiveness of the proposed approach based on metrics.

This research tested the prototype on SutSqlI with known vulnerabilities, and experiments are executed on real-world applications WooCommerce and Drupal. This research compares the experiments with model-based whitebox fuzzing with the results of the experiments with model-based blackbox fuzzing and traditional whitebox fuzzing. The results of these experiments show that model-based whitebox fuzzing is more effective than the other two options. Because the prototype is stateful and uses sequences and targets resources, it has a higher code coverage than the other two fuzzers. It also has a higher chance of valid requests. With more valid requests, the fuzzer tests the deeper parts of the SUT, which only are accessible for valid requests based on previously created objects. Traditional whitebox fuzzing comes close, but it was not able to perform a valid request on Drupal.

6.5. Future work

Future work could focus on different subjects. For example, the following subjects for future work could help to improve the current prototype.

- **More experiments** - The current prototype is tested on two real-world applications. More experiments could provide more data to confirm if model-based whitebox fuzzing is the better option. With the obtained results, model-based whitebox fuzzing looks like the more effective option. However, the results could be different for different SUTs.

- **Coverage guided** - The current prototype builds on the rest-fuzzer project, which is a model-based blackbox fuzzer. This fuzzer makes it possible to generate the code coverage reports when the fuzzer finishes. However, run time code coverage could result in a coverage guided approach. This would enable the fuzzer to target different locations by manipulating the input generator.

- **Different program languages** - The current prototype can only get the code coverage of PHP-based SUTs. Integrating the prototype with different languages would make the prototype more useful for other projects.

- **Analyse source code** - The current prototype uses the OpenAPI specification to generate a model. The OpenAPI specification is popular and available in most languages. However, not all vulnerabilities for REST web services can be discovered while using

the OpenAPI specification. Different types of analysis can be performed to create a fuzzer that can find more vulnerabilities if the source code would be available.

When necessary, the source code for the prototype and the modified SutSqlI project are publically available on Github [1] (prototype[2] and SutSqlI[3]).

[1] https://github.com/
[2] https://github.com/JasonKleuskens1/rest-fuzzer
[3] https://github.com/JasonKleuskens1/SutSqlI

7

REFLECTION

This final section covers a reflection on the process and developed products. These reflections cover some parts of the process that could have been executed in a perhaps better way.

SUT selection During the research preparation, WooCommerce, Magento, PrestaShop and Sylius were selected as SUT. However, from these SUTs, only WooCommerce made it as a SUT. The other SUTs seemed too hard to use due to various reasons. Most were not sharing their OpenAPI specification in a method that would be understandable for a machine. Additionally, all SUTs had different modules to handle REST communication. However, most of them were not editable within the time, which made it hard to track their code coverage within the time.

It would have been good to know this before the testing environments were created. Unfortunately, due to this, Drupal needed to be selected later during the research. This could have been prevented by installing the SUTs before the main phase of this research was started.

Traditional whitebox fuzzing During the second stage of this research, the fuzzer tools needed to be selected to create the prototype. At that time, it was hard to find a whitebox fuzzer that would work with the initially chosen SUTs. At a specific point, a greybox fuzzer was considered to reduce the delay it was causing. Finally, just before the greybox fuzzer was chosen, the PHP Vulnerability fuzzer was found by looking at older tools that were archived.

Looking back at this, selecting both fuzzers in the preparation phase would have been better. If it was not possible to find a whitebox fuzzer, the research questions could have changed before the main phase of the research started.

BIBLIOGRAPHY

[1] Google clusterfuzz. https://google.github.io/clusterfuzz/. Accessed: 2021-03-21. 7

[2] Google oss-fuzz. https://google.github.io/oss-fuzz/. Accessed: 2021-03-21. 7

[3] Microsoft onefuzz. https://www.microsoft.com/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/. Accessed: 2021-03-21. 7

[4] Owasp api security top 10. *apisecurity.io, 2019*. Accessed: 2021-03-22. 2, 23, 24, 25, 27, 28, 29

[5] Owasp top ten 2017. *apisecurity.io, 2017*. Accessed: 2021-06-29. 26

[6] Rest assessment cheat sheet. *OWASP Cheat Sheet Series, 2019*. Accessed: 2021-03-22. 2

[7] V. Atlidakis, P. Godefroid, and M. Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758, 2019. 11, 16, 20, 30, 42

[8] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008. 9

[9] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017. 9

[10] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012. 9

[11] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu. A systematic review of fuzzing techniques. *Computers Security*, 75:118 – 137, 2018. 2, 7, 8, 9, 10

[12] D. Cherry. *Securing SQL server: protecting your database from attackers*. Syngress, 2015. 26

[13] R. T. Fielding. Architectural styles and the design of network-based software architectures. *Doctoral dissertation*, 2000. 6

[14] Fitblip. Suley fuzzer. 9

[15] A. Gerritsen. Model-based fuzzing rest web services to detect vulnerabilities. *Master Thesis*, 1(1):1–98, 2020. 11, 15, 17, 20, 35, 42, 61

[16] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008. 14

[17] P. Godefroid, H. Peleg, and R. Singh. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59. IEEE, 2017. 9

[18] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013. 1, 2

[19] S. Halder. Top 5 api security best practices for 2021. *Postman*. Accessed: 2021-03-22. 23, 24, 25, 27, 28

[20] S. Herbold and A. Hoffmann. Model-based testing as a service, 2017. 11, 37

[21] Itsupportbusiness. Bug vs vulnerability whats difference. *itsb*. Accessed: 2021-06-28. 1

[22] J. Kinder, F. Zuleger, and H. Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 214–228. Springer, 2009. 9

[23] A. Letichevsky, O. Letychevskyi, and V. Peschanenko. Symbolic modelling in white-box model-based testing. In *2015 3rd International Conference on Artificial Intelligence, Modelling and Simulation (AIMS)*, pages 237–240. IEEE, 2015. 12

[24] J. Li, B. Zhao, and C. Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018. 10

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005. 12

[26] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *arXiv preprint arXiv:1812.00140*, 2018. 7, 8, 11

[27] J. Manico. Rest security cheat sheet. *OWASP Cheat Sheet Series, 2019*. 23, 24, 25, 26, 27, 37

[28] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990. 7

[29] S. Mumbaikar, P. Padiya, et al. Web services based on soap and rest principles. *International Journal of Scientific and Research Publications*, 3(5):1–4, 2013. 5

[30] P. Murthy, S. Jell, and A. Ulrich. Model-based testing in industry-a case study with two mbt tools. 2010. 11, 37

[31] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos. The borg: Nanoprobing binaries for buffer overreads. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 87–97, 2015. 9

[32] V.-T. Pham, M. Böhme, and A. Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 543–553, 2016. 3, 12, 13, 15, 16, 22, 30, 33, 35, 40, 41

[33] V.-T. Pham, W. B. Ng, K. Rubinov, and A. Roychoudhury. Hercules: Reproducing crashes in real-world application binaries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 891–901. IEEE, 2015. 14

[34] I. Schieferdecker, J. Grossmann, and M. Schneider. Model-based security testing. *arXiv preprint arXiv:1202.6118*, 2012. 10, 11

[35] M. O. Shudrak and V. V. Zolotarev. Improving fuzzing using software complexity metrics. In *ICISC 2015*, pages 246–261. Springer, 2015. 30

[36] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen. *Fuzzing for software security testing and quality assurance*. Artech House, 2018. 61

[37] H. P. E. Vranken and E. Poll. Software security. *Workbook*, 1(1):16–17, 2019. 1, 2, 31

APPENDIX A: ATTACHED FILES

The directory structure of the supplied files in the folder "Attached files" of the thesis are as follows:

1. **Thesis source files** - The LaTeX source files of this thesis are available within the zip folder: "Thesis - Model-Based Whitebox Fuzzing REST web services.zip".

2. **PHP Vulnerability Hunter** - The source code of PHP Vulnerability Hunter. During the thesis, this tool functioned as a traditional whitebox fuzzer. The source files are available within the zip folder: "PHP Vulnerability Hunter.zip".

3. **Prototype** - The source code of the prototype extending rest-fuzzer. This source code makes it possible to perform different kinds of fuzzing on REST web services, e.g., model-based whitebox fuzzing, model-based blackbox fuzzing and model-based dictionary fuzzing. The source files are available within the zip folder: "Prototype.zip". From this prototype, the model-based whitebox fuzzer and model-based blackbox fuzzer are new (including the frontend).

4. **SutSqlI** - The source code of the modified version of SutSqlI. The source files are available within the zip folder: "SutSqlI.zip".

APPENDIX B: MYSQL QUERIES

Listing 1: Query used to scan for SUTs with the HTTPS vulnerability

```
SELECT proj.* FROM `rmd_suts` sut LEFT JOIN `fuz_projects` proj ON sut
    .id = proj.sut_id WHERE sut.location NOT LIKE '%https://%' AND proj
    .id IN (<project ids>)
```

Listing 2: Query used to scan for SUTs with a potential Access Control vulnerability

```
SELECT * FROM `fuz_responses` WHERE request_id IN ( select id FROM `
    fuz_requests` WHERE project_id IN (<project ids>) and (
    header_parameters_json LIKE '%token%' or  header_parameters_json
    LIKE '%key%')) AND status_code != 401
```

Listing 3: Query used to scan for SUTs with a potential Injection vulnerability

```
SELECT * FROM `fuz_responses` WHERE request_id IN ( select id FROM `
    fuz_requests` WHERE project_id IN (<project ids>) ) AND status_code
     >= 500
```

Listing 4: Query used to scan for SUTs with a potential Validate content types vulnerability

```
SELECT * FROM `fuz_responses` WHERE request_id IN ( select id FROM `
    fuz_requests` WHERE project_id IN (<project ids>) and
    header_parameters_json LIKE '%content-type%' ) AND status_code NOT
    IN (406, 415)
```

Listing 5: Query used to scan for SUTs with a potential Error handling vulnerability

```
SELECT * FROM `fuz_responses` WHERE project_id IN (<project ids>) AND
    status_code >= 500
```

Listing 6: Query used to scan for SUTs with a potential Sensitive information in HTTP requests vulnerability

```
SELECT * FROM `fuz_requests` WHERE project_id IN (<project ids>) AND (
     path_parameters_json LIKE '%key%' OR path_parameters_json LIKE '%
    license%' OR path_parameters_json LIKE '%tax%' OR
    path_parameters_json LIKE '%passport%' OR path_parameters_json LIKE
     '%creditcard%' OR path_parameters_json LIKE '%password%' )
```

APPENDIX C: TEST CONFIGURATIONS

C.1. TRADITIONAL WHITEBOX FUZZER

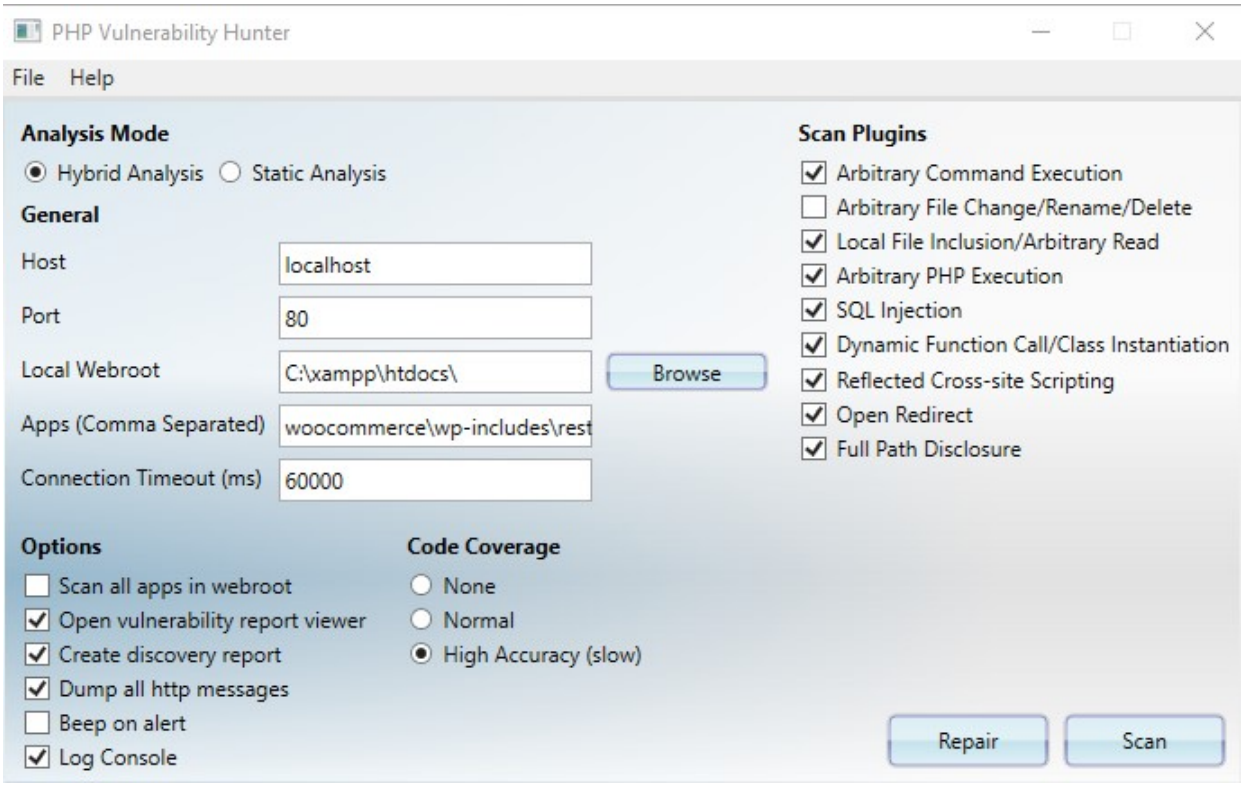

Figure 1: Traditional whitebox fuzzer configuration for the WooCommerce REST web services

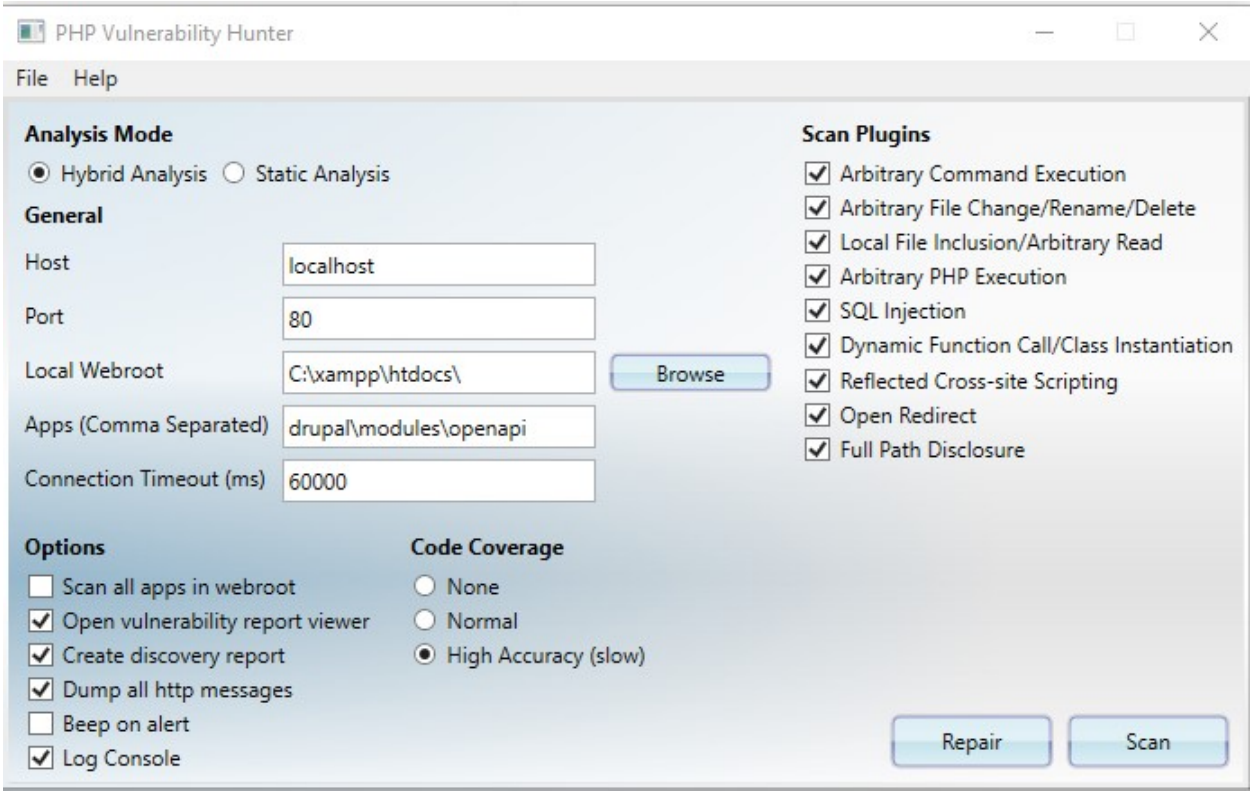

Figure 2: Traditional whitebox fuzzer configuration for the Drupal REST web services

C.2. Model-based blackbox fuzzer

Listing 7: Model-based blackbox fuzzing configuration for the WooCommerce REST web services

```
{
  configurations: {
    "authentication": {
      "method": "BASIC",
      "username": "root",
      "password": "root"
    },
    includeActions: [
      {
        path: ".*",
        httpMethod: ".*"
      }
    ],
    excludeActions: [
      {
        path: "/wp/v2/users/me",
        httpMethod: ".*"
      },
      {
        path: "/wp/v2/users/\\{id\\}",
        httpMethod: "PATCH|POST|PUT"
```

```
      }
    ],
    excludeParameters: [],
    defaults: []
  },
  maxSequenceLength: 1,
  maxNumRequests: 100000,
  duration: 120
}
```

Listing 8: Model-based blackbox fuzzing configuration for the Drupal REST web services

```
{
  configurations: {
    "authentication": {
      "method": "BASIC",
      "username": "root",
      "password": "root"
    },
    includeActions: [
      {
        path: ".*",
        httpMethod: ".*"
      }
    ],
    excludeActions: [
      {
        path: "/wp/v2/users/me",
        httpMethod: ".*"
      },
      {
        path: "/wp/v2/users/\\{id\\}",
        httpMethod: "PATCH|POST|PUT"
      }
    ],
    excludeParameters: [],
    defaults: []
  },
  maxSequenceLength: 1,
  maxNumRequests: 100000,
  duration: 120
}
```

C.3. Model-based whitebox fuzzer

Listing 9: Model-based whitebox fuzzing configuration for the WooCommerce REST web services

```
{
```

```
  configurations: {
    "authentication": {
      "method": "BASIC",
      "username": "root",
      "password": "root"
    },
    includeActions: [
      {
        path: ".*",
        httpMethod: ".*"
      }
    ],
    excludeActions: [],
    excludeParameters: [
      {
        action: {
          path: ".*",
          httpMethod: ".*"
        },
        parameter: {
          name: "template|meta|subtype|status|username|roles
             |parent",
          required: "false"
        }
      }
    ],
    defaults: [
      {
        action: {
          path: ".*",
          httpMethod: ".*"
        },
        parameter: {
          name: ".*",
          required: ".*"
        },
        default: ""
      }
    ]
  },
  maxSequenceLength: 1,
  maxNumRequests: 100000,
  duration: 120,
  thresholdRequiredNonFuzzableMinimumRequests: "1",
  thresholdRequiredNonFuzzableValue: "1"
}
```

Listing 10: Model-based whitebox fuzzing configuration for the Drupal REST web services

```
{
  configuration: {
    authentication: {
      method: "BASIC",
      username: "root",
      password: "root"
    },
    includeActions: [
      {
        path: ".*",
        httpMethod: ".*"
      }
    ],
    excludeActions: [],
    excludeParameters: [
      {
        action: {
          path: ".*",
          httpMethod: ".*"
        },
        parameter: {
          name: "template|meta|subtype|status|username|roles
             |parent",
          required: "false"
        }
      }
    ],
    defaults: [
      {
        action: {
          path: ".*",
          httpMethod: ".*"
        },
        parameter: {
          name: ".*",
          required: ".*"
        },
        default: ""
      }
    ]
  },
  maxSequenceLength: 3,
  maxNumRequests: 100000,
  duration: 120,
  thresholdRequiredNonFuzzableMinimumRequests: "1",
```

```
45   thresholdRequiredNonFuzzableValue: "1"
46 }
```